A reliable Turing machine

Ilir Çapuni Peter Gács

January 23, 2014

Abstract

The title says it.

Contents

Contents

1 Introduction

1.1 To be written

1.2 Turing machines

Our contribution uses one of the standard definitions of a Turing machine.

A Turing machine M is defined by a tuple

$$(\Gamma, \Sigma, \delta, q_{\text{start}}, F).$$

Here, Γ is a finite set of ***states***, Σ is a finite alphabet, and

$$\delta\colon \Sigma \times \Gamma \to \Sigma \times \Gamma \times \{-1, 0, +1\}$$

is the transition function. The tape alphabet Σ contains at least the distinguished symbols $␣, 0, 1$ where $␣$ is called the ***blank symbol***. The state q_{start} is called the ***starting state***, and there is a set F of ***final states***.

The tape is blank at all but finitely many positions.

A ***configuration*** is a tuple

$$(q, A, h),$$

where $q \in \Gamma$ is the ***current state***, $h \in \mathbb{Z}$ is the current ***head position***, or ***observed cell***, and $A \in \Sigma^{\mathbb{Z}}$ is the ***tape content***: at position p, the tape contains the symbol $A[p]$. If $C = (q, A, h)$ is a configuration then we will write

$$C.\text{state} = q, \quad C.\text{pos} = h, \quad C.\text{tape} = A.$$

Here, A is also called the ***tape configuration***. The cell at position h is the ***current cell***. Though the tape alphabet may contain non-binary symbols, we will restrict input and output to binary.

The transition function δ tells us how to compute the next configuration from the present one. When the machine is in a state q, at tape position h, and observes tape cell with content a, then denoting

$$(a', q', j) = \delta(a, q),$$

it will change the state to q', change the tape content at position h to a', and move to tape position to $h + j$. For $q \in F$ we have $a' = a$, $q' \in F$.

Definition 1.1 (Fault)**.** We say that a ***fault*** occurs at time t if the output (a', q', j) of the transition function at this time is replaced with some other value (which is then used to compute the next configuration). ⌟

1.3 Codes, and the result

For fault-tolerant computation, some redundant coding of the information is needed.

Definition 1.2 (Codes)**.** Let Σ_1, Σ_2 be two finite alphabets. A ***block code*** is given by a positive integer Q—called the ***block size***—and a pair of functions

$$\psi_* : \Sigma_2 \to \Sigma_1^Q, \quad \psi^* : \Sigma_1^Q \to \Sigma_2$$

with the property $\psi^*(\psi_*(x)) = x$. It is extended to strings by encoding each letter individually: $\psi_*(x_1, \ldots, x_n) = \psi_*(x_1) \cdots \psi_*(x_n)$. ⌟

For ease of spelling out a result, let us consider only computations whose outcome is a single symbol, at tape position 0.

Theorem 1. *There is a Turing machine M_1 with a function $a \mapsto a.output$ defined on its alphabet, such that for any Turing machine G with alphabet Σ and state set Γ there are $0 \le \varepsilon < 1$ and $\alpha_1, \alpha_2 > 0$ with the following property.*

For each input length $n = |x|$ a block code (φ_*, φ^*) of block size $Q = O((\log n)^{\alpha_1})$ can be constructed such that the following holds.

Let M_1 start its work from its initial state, and the initial tape configuration $\xi = (q_{\text{start}}, \varphi_*(x), 0)$. Assume further that during its operation, faults occur independently at random with probabilities $\le \varepsilon$.

Suppose that on input x machine G reaches a final state at time t and writes value y at position 0 of the tape. Then denoting by $\eta(u)$ the configuration machine M_1 at time u, at any time t' after

$$t \cdot (\log t)^{\alpha_2},$$

we have $\eta(t').\text{tape}[0].output = y$ with probability at least $1 - O(\varepsilon)$.

We emphasize that the actual code φ of the construction will depend on n only in a simple way: it will be the "concatenation" of one and the same fixed-size code with itself, $O(\log \log n)$ times.

2 Overview of the construction

A Turing machine that simulates "reliably" any other Turing machine even when it is subjected to isolated bursts of faults of constant size, is given in [1]. By ***reliably*** we mean that the simulated computation can be decoded from the history of the simulating machine despite occasional damages.

2.1 Isolated bursts of faults

Let us give a brief overview of a machine M_1 that can withstand isolated bursts of faults, as most of its construction will be reused in the probabilistic setting.

Let us break up the task of error correction into several problems to be solved. The solution of one problem gives rise to another problem one, but the process converges.

Redundant information. The tape information of the simulated Turing machine will be stored in a redundant form, more precisely in the form of a block code.

Redundant processing. The block code will be decoded, the retrieved information will be processed, and the result recorded. To carry out all this in a way that limits the propagation of faults, the tape will be split into tracks that can be handled separately, and the major processing steps will be carried out three times within one work period.

Local repair. All the above process must be able to recover from a local burst of faults. For this, it is organized into a rigid, locally checkable structure with the help of local addresses, and some other tools like sweeps and short switchbacks (zigzags). The major tool of local correction, the local healing procedure, turns out to be the most complex part of the construction.

Disturbed local repair. A careful organization of the healing procedure makes sure that even if a new burst interrupts it (or jumps into its middle), soon one or two new invocations of it will finish the job (whenever needed).

Here is some more detail.

Each tape cell of the simulated machine M_2 will be represented by a block of size Q of the simulating machine M_1 called a ***colony***. Each step of M_2 will be simulated by a computation of M_1 called a ***work period***. During this time, the head of M_1 makes a number of sweeps over the current colony, decodes the represented cell symbol and state, computes the new state, and transfers the necessary information to the neighbor colony that corresponds to the new position of the head of M_2.

In order to protect information from the propagation of errors, the tape of M_1 is subdivided into ***tracks***: each track corresponds to a ***field*** of a cell symbol of M_1 viewed as a data record. Each stage of computation will be repeated three times. The results will be stored in separate tracks, and a final cell-by-cell majority vote will recover the result of the work period from them.

All this organization is controlled by a few key fields, for example a field called *cAddr* showing the position of each cell in the colony, and a field *cSw*

showing the last sweep of the computation (along with its direction) that has been performed already. The technically most challenging part of the construction is the protection of this control information from bursts.

For example, a burst can reverse the head in the middle of a sweep. Our goal is that such structural disruptions be discovered locally, so we cannot allow the head to go far from the place where it was turned back. Therefore the head's movement will not be straight even during a single sweep: it will make frequent zigzags. This will trigger alarm, and the start of a healing procedure if for example a turn-back is detected.

It is a significant challenge that the healing procedure itself can be interrupted (or started) by a burst.

2.2 Hierarchical construction

In order to build a machine that can resist faults occurring independently of each other with some small probability, we take the approach suggested in [5], and implemented in [3] and [4] for the case of one-dimensional cellular automata, with some ideas from the tiling application of [2]. We will build a ***hierarchy of simulations***: machine M_1 simulates machine M_2 which simulates machine M_3, and so on. For simplicity we assume all these machines have the same program, and all simulations have the same block size.

One cell of machine M_3 is simulated by one colony of machine M_2. Correspondingly, one cell of M_2 is simulated by one colony of machine M_1. So one cell of M_3 is simulated by Q^2 cells of M_1. Further, one step of machine M_3 is simulated by one work period of M_2 of, say, $O(Q^2)$ steps. One step of M_2 is simulated by one work period of M_1, so one step of M_3 is simulated by $O(Q^4)$ steps of M_1.

Per construction, machine M_1 can withstand bursts of faults whose size is $\leq \beta$ for some constant parameter β, that are separated by some $O(Q^2)$ fault-free steps. Machines M_2, M_3, ... have the same program, so it would be natural to expect that machine M_1 can withstand also some *additional*, larger bursts of size $\leq \beta Q$ if those are separated by at least $O(Q^4)$ steps.

But a new obstacle arises. On the first level, damage caused by a big burst spans several colonies. The repair mechanism of machine M_1 outlined in Section 2.1 above is too local to recover from such extensive damage. This cannot be allowed, since then the whole hierarchy would stop working. So we add a new mechanism to M_1 that, more modestly, will just try to restore a large enough portion of the tape, so it can go on with the simulation of M_2, even if

all original information was lost. For this, M_1 may need to rewrite an area as large as a few colonies.

This will enable the low-level healing procedure of machine M_2 to restore eventually a higher-level order.

All machines above M_1 in the hierarchy are "virtual": the only hardware in the construction is machine M_1.

A tricky issue is "forced self-simulation": while we are constructing machine M_1 we want to give it the feature that it will simulate a machine M_2 that works just like M_1. The "forced" feature means that this simulation should work without any written program (that could be corrupted).

This will be achieved by a construction similar to the proof of the Kleene's fixed-point theorem (also called recursion theorem). We first fix a (simple) programming language to express the transition function of a Turing machine. We write an interpreter for it in this same language (just as compilers for the C language are sometimes written in C). The program of the transition function of M_2 (essentially the same as that of M_1) in this language, is a string that will be "hard-wired" into the transition function of M_1, so that M_1, at the start of each work period, can write it on a working track of the base colony. Then the work period will interpret it, applying it to the data found there, resulting in the simulation of M_2.

In this way, an infinite sequence of simulations arises, in order to withstand larger and larger but sparser and sparser bursts of faults.

Since the M_1 uses the universal interpreter, which in turns simulates the same program, it is natural to ask how machine M_1 simulates a given Turing machine G that does the actual useful computation? For this task, we set aside a separate track on each machine M_i, on which some arbitrary other Turing machine can be simulated. The higher the level of the machine M_k that performs this "side-simulation", the higher the reliability. Thus, only the simulations $M_k \to M_{k+1}$ are forced, without program (that is a hard-wired program): the side simulations can rely on written programs, since the firm structure in the hierarchy $M_1, M_2, \ldots$ will support them reliably.

2.3 From combinatorial to probabilistic noise

The construction we gave in the previous subsection was related to increasing bursts that are not frequent. In essence, that noise model is combinatorial. To deal with probabilistic noise combinatorially, we stratify the set of faulty times *Noise* as follows. For a series of parameters β_k, V_k, we first remove "isolated

bursts" of type (β_1, V_1) of elements of this set. (The notion of "isolated bursts" of type (β, V) will be defined appropriately.) Then, we remove isolated bursts of type (β_2, V_2) from the remaining set, and so on. It will be shown that with the appropriate choice of parameters, with probability 1, eventually nothing is left over from the set *Noise*.

A composition of two reliable simulations is even more reliable. We will see that a sufficiently large hierarchy of such simulations resists probabilistic noise.

2.4 Difficulties

Let us spell-out some of the main problems that the paper deals with, and some general ways they will be solved or avoided.

- A large burst of M_1 can modify the order of entire colonies or create new ones with gaps between them.

  To overcome this problem conceptually, we introduce the notion of a ***generalized Turing machine*** allowing for non-adjacent cells. Each such machine has a parameter B called the ***cell body size***. The cell body size of a Turing machine in Section 1.2 would still remain 1.

- What to do when the head is in a middle of an empty area where no structure exists? To ensure reliable passage across such areas, we will try to keep everything filled with cells, even if these are not part of the main computation.

- Noise can create areas over which the predictability of the simulated machine is limited. In these islands the (on this level) invisible structure of the underlying simulation may be destroyed. These areas should not simply be considered blank, since blankness implies predictable behavior. They will be called ***damaged***. When the head is in or near damage then even in the absence of new faults, the predictability of the behavior of the (simulated) machine will be severely limited.

- Due to limited predictability over damage, the definition of the generalized Turing machine stipulates a certain "magical" erasure of damage in case the head stays long enough on it. (Of course, this property needs to be implemented in simulation, which is one of the main burdens of the actual construction.) Once damage is erased the area will be re-populated with new cells. Their content is not important, what matters is the restoration of predictability.

- If local repair fails, a special rule will be invoked that reorganizes a larger part of the tape (of the size of a few colonies instead of only a few cells). This is the mechanism enabling the "magical" restoration on the next level.

2.5 A shortcut solution

A fault-tolerant one-dimensional cellular automaton is constructed in [4]. If our Turing machine could just simulate such an automaton, it would become fault-tolerant. This can indeed almost be done provided that the size of the computation is known in advance. The cellular automaton can be made finite, and we could define a "kind of" Turing machine with a *circular tape* simulating it. But this solution requires input size-dependent hardware.

It seems difficult to define a fault-tolerant sweeping behavior on a regular Turing machine needed to simulate cellular automaton, without recreating an entire hierarchical construction—as we are doing here.

3 Notation

Most notational conventions given here are common; some other ones will also be useful.

Natural numbers and integers. By $\mathbb{Z}$ we denote the set of integers.

$$\mathbb{Z}_{>0} = \{x : x \in \mathbb{Z},\ x{>}0\},$$
$$\mathbb{Z}_{\geqslant 0} = \mathbb{N} = \{x : x \in \mathbb{Z},\ x \geqslant 0\}.$$

Intervals. We use the standard notation for intervals:

$$[a,b] = \{x : a \leq x \leq b\}, \quad [a,b) = \{x : a \leq x{<}b\},$$
$$(a,b] = \{x : a{<}x \leq b\}, \quad (a,b) = \{x : a{<}x{<}b\}.$$

We will also write $[a,b)$ in place of $[a,b) \cap \mathbb{Z}$, whenever this leads to no confusion. Instead of $[x+a, x+b)$, sometimes we will write

$$x + [a,b).$$

Ordered pairs. Ordered pairs are also denoted by (a,b), but it will be clear from the context whether we are referring to an ordered pair or open interval.

Comparing the order of a number and an interval. For a given number x and interval I, we write

$$x \geqslant I$$

if for every $y \in I$, $x \geqslant y$.

Distance. The distance between two real numbers x and y is defined in a usual way:

$$d(x,y) = |x - y|.$$

The ***distance of a point*** x ***from interval*** I is

$$d(x,I) = \min_{y \in I} d(x,y).$$

Ball, neighborhood, ring, stripe. A ***ball of radius*** $r>0$***, centered at*** x is

$$B(x,r) = \{y : d(x,y) \leq r\}.$$

An r***-neighborhood of interval*** I is

$$\{x : d(x,I) \leq r\}.$$

An r***-ring*** around interval I is

$$\{x : d(x,I) \leq r \text{ and } x \notin I\}.$$

A r***-stripe to the right of interval*** I is

$$\{x : d(x,I) \leq r \text{ and } x \notin I \text{ and } x>I\}.$$

Logarithms. Unless specified differently, the base of logarithms throughout this work is 2.

4 Describing a Turing machine

Let us introduce the tools allowing to describe the reliable Turing machine.

4.1 Universal Turing machine

We will describe our construction in terms of universal Turing machines, operating on binary strings as inputs and outputs. We define universal Turing machines in a way that allows for rather general "programs".

Definition 4.1 (Standard pairing)**.** For a (possibly empty) binary string $x = x(1)\cdots x(n)$ let us introduce the map

$$\langle x\rangle = 0^{|x|}1x,$$

Now we encode pairs, triples, and so on, of binary strings as follows:

$$\langle s,t\rangle = \langle s\rangle t,$$
$$\langle s,t,u\rangle = \langle\langle s,t\rangle,u\rangle,$$

and so on.

From now on, we will assume that our alphabets Σ, Γ are of the form $\Sigma = \{0,1\}^s$, $\Gamma = \{0,1\}^g$, that is our tape symbols and machine states are viewed as binary strings of a certain length. Also, if we write $\langle i,u\rangle$ where i is some number, it is understood that the number i is represented in a standard way by a binary string. ⌟

Definition 4.2 (Computation result, universal machine)**.** Assume that a Turing machine M starting on binary x, at some time t arrives at the first time at some final state. Then we look at the longest (possibly empty) binary string to be found starting at position 0 on the tape, and call it the ***computation result*** $M(x)$. We will write

$$M(x,y) = M(\langle x,y\rangle),\quad M(x,y,z) = M(\langle x,y,z\rangle),$$

and so on.

A Turing machine U is called ***universal*** among Turing machines with binary inputs and outputs, if for every Turing machine M, there is a binary string p_M such that for all x we have $U(p_M,x) = M(x)$. (This equality also means that the computation denoted on the left-hand side reaches a final state if and only if the computation on the right-hand side does.) ⌟

Let us introduce a special kind of universal Turing machines, to be used in expressing the transition functions of other Turing machines. These are just the Turing machines for which the so-called s_{mn} theorem of recursion theory holds with $s(x,y) = \langle x,y\rangle$.

Definition 4.3 (Flexible universal Turing machine)**.** A universal Turing machine will be called ***flexible*** if whenever p has the form $p = \langle p', p'' \rangle$ then

$$U(p, x) = U(p', \langle p'', x \rangle).$$

Even if x has the form $x = \langle x', x'' \rangle$, this definition chooses $U(p', \langle p'', x \rangle)$ over $U(\langle p, x' \rangle, x'')$, that is starts with parsing the first argument (this process converges, since x is shorter than $\langle x, y \rangle$). ⌟

It is easy to see that there are flexible universal Turing machines. On input $\langle p, x \rangle$, a flexible machine first checks whether its "program" p has the form $p = \langle p', p'' \rangle$. If yes, then it applies p' to the pair $\langle p'', x \rangle$. (Otherwise it just applies p to x.)

Definition 4.4 (Transition program)**.** Consider an arbitrary Turing machine M with state set Γ, alphabet Σ, and transition function δ. A binary string π will be called a ***transition program*** of M if whenever $\delta(a, q) = (a', q', j)$ we have

$$U(\pi, a, q) = \langle a', q', j \rangle.$$

We will also require that the computation induced by the program makes $O(|p| + |a| + |q|)$ left-right turns, over a length tape $O(|p| + |a| + |q|)$. ⌟

The transition program just provides a way to compute the (local) transition function of M by the universal machine, it does not organize the rest of the simulation.

Remark 4.5. In the construction of universal Turing machines provided by the textbooks (though not in the original one given by Turing), the program is generally a string encoding a table for the transition function δ of the simulated machine M. Other types of program are imaginable: some simple transition functions can have much simpler programs. However, our fixed machine is good enough (similarly to the optimal machine for Kolmogorov complexity). If some machine U' simulates M via a very simple program q, then

$$M(x) = U'(q, x) = U(p_{U'}, \langle q, x \rangle) = U(\langle p_{U'}, q \rangle, x),$$

so U simulates this computation via the program $\langle p_{U'}, q \rangle$. ⌟

4.2 Rule language

In what follows we will describe the generalized Turing machines M_k for all k. They are all similar, differing only in the parameter k; the most important activity of M_k is to simulate M_{k+1}. The description will be uniform, except for the parameter k. We will denote therefore M_k simply by M, and M_{k+1} by M^*. Similarly we will denote the block size Q_k of the block code of the simulation simply by Q.

Instead of writing a huge table describing the transition function $\delta_k = \delta_„$ we present the transition function as a set of ***rules***. It will be then possible to write one *interpreter* program that carries out these rules; that program can be written for some fixed flexible universal machine Univ.

Each rule consists of some (nested) conditional statements, similar to the ones seen in an ordinary program: "**if** *condition* **then** *instruction* **else** *instruction*", where the condition is testing values of some fields of the state and the observed cell, and the instruction can either be elementary, or itself a conditional statement. The elementary instructions are an ***assignment*** of a value to a field of the state or cell symbol, or a command to move the head. Rules can call other rules, but these calls will never form a cycle. Calling other rules is just a shorthand for nested conditions.

Even though rules are written like procedures of a program, they describe a single transition. When several consecutive statements are given, then they change different fields of the state or cell symbol, so they can be executed simultaneously.

Assignment of value x to a field y of the state or cell symbol will be denoted by $y \leftarrow x$. We will also use some conventions introduced by the C language: namely, $x \leftarrow x+1$ and $x \leftarrow x-1$ are abbreviated to $x{+}{+}$ and $x{-}{-}$ respectively.

Rules can also have parameters, like $\texttt{Swing}(a,b,u,v)$. Since each rule is called only a constant number of times in the whole program, the parametrized rule can be simply seen as a shorthand.

Mostly we will describe the rules using plain English, but it should always be clear that they are translatable into such rules.

For the machine M we are constructing, each state will be a tuple $q = (q_1, q_2, \ldots, q_k)$, where the individual elements of the tuple will be called ***fields***, and will have symbolic names. For example, we will have fields *Addr* and *Drift*, and may write q_1 as $q.Addr$ or just *Addr*, q_2 as $q.Drift$ or *Drift*, and so on.

Similarly for tape symbols. In order to distinguish fields of tape symbols from fields of the state, we will always start the name of a field of the tape

symbols by the letter c. We have seen already one example of this, the field *cDir* of tape symbols in the definition of a generalized Turing machine.

In what follows we describe some of the most important fields we will use; others will be introduced later.

A properly formatted configuration of M splits the tape into blocks of Q consecutive cells called ***colonies***. One colony of the tape of the simulating machine represents one cell of the simulated machine. The colony that corresponds to the cell that the simulated machine is scanning is called the ***base colony*** (a precise definition will be based on the actual history of the work of M). Once the drift is known, the union of the base colony with the target colony in the direction of the drift is called the ***workspace*** (this notion will need to be defined more carefully later).

There will be a field of the state called the ***mode***:

$$Mode \in \{\text{Normal}, \text{Healing}, \text{Rebuilding}\}.$$

In the ***normal*** mode, the the machine is engaged in the regular business of simulation. The ***healing*** mode tries to correct some local fault due to a couple of neighboring bursts, while the ***rebuilding*** mode attempts to restore the colony structure on the scale of a couple of colonies.

The content of each cell of the tape of M also has several fields. Some of these have names identical to fields of the state. In describing the transition rule of M we will write, for example, $q.Addr$ simply as $Addr$, and for the corresponding field of the observed cell symbol a we will write $a.cInfo$, or just $cInfo$. The array of values of the same field of the cells will be called a ***track***. Thus, we will talk about the *cHold* track of the tape, corresponding to the *cHold* field of cells.

Each field of a cell has also a possible value $\emptyset$ whose approximate meaning is "undefined".

Some fields and parameters are important enough to introduce them right away. The

$$cInfo, cState$$

track of a colony of M contain the strings that encode the content of the simulated cell of M^* and its simulated state respectively.

$$cProg$$

track stores the program of M^*, in an appropriate form to be interpreted by the simulation. The field

$$cAddr$$

of the cell shows the position of the cell in its colony: it takes values in $[-Q, 2Q)$, since the addresses in a bridge (see later) will be continuations of those in the colony (which run from 0 to $Q-1$). The colony along with the adjacent cells that continue its addresses will be called an ***extended colony***. A colony can only be extended in the direction of the drift of its cells. There is a corresponding *Addr* field of the state. The

$$Sw$$

field counts the sweeps that the head makes during the work period. There is a corresponding *cSw* field in the cell. The direction $d \in \{-1, 0, 1\}$ in which the simulated head moves will be denoted by

$$Drift.$$

There is a corresponding field *cDrift*. The number of the last sweep of the work period will depend on the drift d, and will be denoted by

$$\text{Last}(d). \tag{4.1}$$

Cells will be designated as belonging to a number of possible ***kinds***, signaled by the field

$$cKind$$

with values

$$\text{Member}, \text{Target}, \text{Vac}, \text{Stem}.$$

Here is a description of the role of these cell kinds. Normally, cells will have the kind Member. During the simulation, however, the elements of the colony that is to become the next base colony, will be made to have the kind Target. If the neighbor colony is not adjacent, then the base colony will be extended to it by a ***bridge***. Bridge cells are member cells with addresses outside $[0, Q)$. Though they have the kind Member, they will frequently be treated differently from the other member cells.

The stem kind is sometimes convenient when some cells need to be created temporarily that do not participate in any known colony structure. We will also try to keep all areas between colonies filled with (not necessarily adjacent) stem cells.

During healing, some special fields of the state and cell are used, they will be subfields of the fields

$$Heal \tag{4.2}$$

and *cHeal* respectively. In particular, there will be *Heal.Sw* and *cHeal.Sw* fields. We will say that a cell is ***marked for healing*** if $cHeal.Sw \neq 0$. Similarly, during rebuilding we will work with subfields of the field *Rebd* and *cRebd*, and a cell will be called ***marked for rebuilding*** if $cRebd.Sw \neq 0$.

5 Exploiting structure in the noise

5.1 Sparsity

Let us introduce a technique connecting the combinatorial and probabilistic noise models.

Definition 5.1 (Centered rectangles, isolation)**.** Let $\boldsymbol{r} = (r_1, r_2)$, $r_1, r_2 \geqslant 0$, be a two-dimensional nonnegative vector. An ***rectangle*** of radius $\boldsymbol{r}$ ***centered*** at $\boldsymbol{x}$ is

$$B(\boldsymbol{x}, \boldsymbol{r}) = \{\boldsymbol{y} : |y_i - x_i| \leq r_i, i = 1, 2\}. \tag{5.1}$$

Let $E \subseteq \mathbb{Z}^2$ be a two-dimensional set. A point $\boldsymbol{x}$ of E is $(\boldsymbol{r}, \boldsymbol{r}^*)$***-isolated*** if

$$E \cap B(\boldsymbol{x}, \boldsymbol{r}^*) \subseteq B(\boldsymbol{x}, \boldsymbol{r}).$$

Let

$$D(E, \boldsymbol{r}, \boldsymbol{r}^*) = \{\boldsymbol{x} \in E : \boldsymbol{x} \text{ is not } (\boldsymbol{r}, \boldsymbol{r}^*)\text{-isolated from } E\}. \tag{5.2}$$

⌟

Definition 5.2 (Sparsity)**.** Let $\beta \geqslant 9$ be a constant, and let $0<B_1<B_2<\cdots$, $T_1<T_2<\cdots$ be sequences of positive integers to be fixed later.

For a two-dimensional set E, let $E^{(1)} = E$. For $k>1$ we define recursively:

$$E^{(k+1)} = D(E^{(k)}, \beta(B_k, T_k), (B_{k+1}, T_{k+1})). \tag{5.3}$$

Set $E^{(k)}$ is called the ***k-th residue*** of E.

Set E is $(\boldsymbol{r},\boldsymbol{r}^*)$-***sparse*** if $D(E,\boldsymbol{r},\boldsymbol{r}^*)=\emptyset$, that is it consists of $(\boldsymbol{r},\boldsymbol{r}^*)$-isolated points. It is k-***sparse*** if $E^{(k+1)}=\emptyset$. It is simply ***sparse*** if $\bigcap_k E^{(k)}=\emptyset$.

When $E=E^{(k)}$ and k is know then we will denote $E^{(k+1)}$ simply by E^*. ⌟

The following lemma connects the above defined sparsity notions to the requirement of small fault probability.

Lemma 5.3 (Sparsity). *Let $1=B_1\le B_2\le\cdots$ and $1=T_1<T_2<\cdots$ be sequences of integers with $Q_k=B_{k+1}/B_k$, $U_k=T_{k+1}/T_k$, and*

$$\lim_{k\to\infty}\frac{\log(U_kQ_k)}{1.5^k}=0, \tag{5.4}$$

For sufficiently small ε, for every $k\geqslant 1$ the following holds. Let $E\subseteq\mathbb{Z}\times\mathbb{Z}_{\geqslant 0}$ be a random set with the property that each pair (p,t) belongs to E independently from the other ones with probability $\le\varepsilon$.

Then for each point $\boldsymbol{x}$ and each k,

$$\mathbb{P}\{B(\boldsymbol{x},(B_k,T_k))\cap E^{(k)}\ne\emptyset\}<2\varepsilon\cdot 2^{-1.5^k}.$$

As a consequence, the set E is sparse with probability 1.

Proof. Let $k=1$. Rectangle $B(\boldsymbol{x},(B_1,T_1))$ is a single point, hence the probability of our event is $<\varepsilon$. Let us prove the inequality by induction, for $k+1$.

Note that our event depends at most on the rectangle $B(\boldsymbol{x},3(B_k,T_k))$. Let

$$p_k=2\varepsilon\cdot 2^{-1.5^k}.$$

Suppose $\boldsymbol{y}\in E^{(k)}\cap B(\boldsymbol{x},(B_{k+1},T_{k+1}))$. Then, according to the definition of $E^{(k)}$, there is a point

$$\boldsymbol{z}\in B(\boldsymbol{y},T_{k+1})\cap E^{(k)}\setminus B(\boldsymbol{y},\beta(B_k,T_k)). \tag{5.5}$$

Consider a standard partition of the (two-dimensional) space-time into rectangles $K_p=\boldsymbol{c}_p+[-B_k,B_k)\times[-T_k,T_k)$ with centers $\boldsymbol{c}_1,\boldsymbol{c}_2,\ldots$. The rectangles K_i,K_j containing $\boldsymbol{y}$ and $\boldsymbol{z}$ respectively intersect $B(\boldsymbol{x},2(B_{k+1},T_{k+1}))$. The triple-size rectangles $K_i'=c_i+[-3B_k,3B_k)\times[-3T_k,3T_k)$ and K_j' are disjoint, since (5.5) implies $|y_1-z_1|>\beta B_k$ and $|y_2-z_2|>\beta T_k$.

The set $E^{(k)}$ must intersect two rectangles K_i, K_j of size $2(B_k,T_k)$ separated by at least $4(B_k,T_k)$, of the big rectangle $B(\boldsymbol{x},2(B_{k+1},T_{k+1}))$.

By the inductive hypothesis, the event $\mathcal{F}_i$ that K_i intersects E_k has probability bound p_k. It is independent of the event $\mathcal{F}_j$, since these events depend only on the triple size disjoint rectangles K_i' and K_j'.

The probability that both of these events hold is at most p_k^2. The number of possible rectangles K_p intersecting $B(\boldsymbol{x}, 2(B_{k+1}, T_{k+1}))$ is at most $C_k := ((2U_kQ_k)+2)^2$, so the number of possible pairs of rectangles is at most $C_k^2/2$, bounding the probability of our event by

$$\begin{aligned} C_k^2 p_k^2/2 &= 2C_k^2\varepsilon^2 2^{-1.5^{k+1}} \cdot 2^{-0.5\cdot 1.5^k} \\ &= 2\varepsilon 2^{-1.5^{k+1}} \cdot \varepsilon C_k^2 2^{-0.5\cdot 1.5^k}. \end{aligned}$$

Since $\lim_k \frac{\log(U_kQ_k)}{1.5^k} = 0$, the last factor is ≤ 1 for sufficiently small ε. □

5.2 Error-correcting code

Let us add error-correcting features to block codes introduced in Definition 1.2.

Definition 5.4 (Error-correcting code)**.** A block code is (β, t)-***burst-error-correcting***, if for all $x \in \Sigma_2$, $y \in \Sigma_1^Q$ we have $\psi^*(y) = x$ whenever y differs from $\psi_*(x)$ in at most t intervals of size $\leq \beta$. ⌟

Example 5.5 (Repetition code)**.** Suppose that $Q \geqslant 3\beta$ is divisible by 3, $\Sigma_2 = \Sigma_1^{Q/3}$, $\psi_*(x) = xxx$. Let $\psi^*(y)$ be obtained as follows. If $y = y(1)\dots y(Q)$, then $x = \psi^*(y)$ is defined as follows: $x(i) = \mathrm{maj}(y(i), y(i + Q/3), y + 2Q/3)$. For all $\beta \leq Q/3$, this is a $(\beta, 1)$-burst-error-correcting code.

If we repeat 5 times instead of 3, we get a $(\beta, 2)$-burst-error-correcting code. Let us note that there are much more efficient such codes than just repetition. ⌟

Let us assume that a generalized Turing machine

$$M = (\Gamma, \Sigma, \delta, \mathit{NonAdj}, \mathit{cDir}, q_{\mathrm{start}}, F, B, T)$$

is used to simulate a generalized Turing machine

$$M^* = (\Gamma^*, \Sigma^*, \delta^*, \mathit{NonAdj}^*, \mathit{cDir}^*, q^*_{\mathrm{start}}, F^*, B^*, T^*).$$

We will assume that $\Gamma^* \cup \{\emptyset\}$, and the alphabet Σ^* are subsets of the set of binary strings $\{0,1\}^\ell$ for some $\ell < Q$ (we can always ignore some states or tape symbols, if we want).

We will store the coded information at distance

$$PadLen \tag{5.6}$$

from the end of our colony of size Q. So let (υ_*, υ^*) be a $(\beta, 2)$-burst-error-correcting block code

$$\upsilon_* : \{0,1\}^\ell \cup \{\emptyset\} \to \{0,1\}^{(Q-2\cdot PadLen)B}.$$

We could use, for example, the repetition code of Example 5.5. Other codes are also appropriate, but we require that they have some fixed programs p_{encode}, p_{decode} on the universal machine Univ, in the following sense:

$$\upsilon_*(x) = \text{Univ}(p_{\text{encode}}, x), \quad \upsilon^*(y) = \text{Univ}(p_{\text{decode}}, y).$$

Also, these programs must work in quadratic time and linear space on a one-tape Turing machine (as the repetition code certainly does).

Let us now define the block code (ψ_*, ψ^*) used in the definition of the configuration code (φ_*, φ^*) as outlined in Section 6.3. We define

$$\psi_*(a) = 0^{PadLen}\upsilon_*(a)0^{PadLen}. \tag{5.7}$$

It will be easy to compute the configuration code from ψ_*, once we know what fields there are which ones need initialization.

The decoded value $\psi^*(x)$ is obtained by first removing *PadLen* symbols from both ends of x to get x', and then computing $\upsilon^*(x')$.

6 The model

6.1 Generalized Turing machine

Standard Turing machines do not have operations like "creation" or "killing" of cells, nor do they allow for cells to be non-adjacent. We introduce here a ***generalized Turing machine***. It depends on an integer $B \geqslant 1$ that denotes the cell body size, and an upper bound T on the transition time. These parameters are convenient since they provide the illusion that the different Turing machines in the hierarchy of simulations all operate on the same linear space. Even if the notions of cells, alphabet and state are different for each machine of the hierarchy, at least the notion of a *location on the tape* is the same.

Definition 6.1 (Generalized Turing machine)**.** A ***generalized Turing machine*** M is defined by a tuple

$$(\Gamma, \Sigma, \delta, NonAdj, cDir, q_{\text{start}}, F, B, T), \tag{6.1}$$

where Γ and Σ are finite sets called the ***set of states*** and the ***alphabet*** respectively,

$$\delta : \Sigma \times \Gamma \to \Sigma \times \Gamma \times \{-1, 0, 1\},$$

is the ***transition function***. The function ("field") $NonAdj\colon \Gamma \to \{\text{true}, \text{false}\}$ of the state will show whether the last move was from a non-adjacent cell. The function ("field") $cDir\colon \Sigma \to \{-1, 0, 1\}$ of the cell content needs to always point towards the head, so the transition function δ is required to have the property that if $(a', q', j) = \delta(a, q)$ then $a'.cDir = j$.

The role of starting state q_{start} and final states in F is as before. The integer $B \geqslant 1$ is called the ***cell body size***, and the real number T is a bound on the transition time.

Among the elements of the tape alphabet Σ, we distinguish the elements $0, 1, Bad, \text{Vac}$. The role of the symbols Bad and Vac will be clarified below. ⌟

Definition 6.2 (Configuration)**.** Consider a generalized Turing machine (6.1). A ***configuration*** is a tuple

$$(q, A, h, \tilde{h}),$$

where $q \in \Gamma$, $h, \tilde{h} \in \mathbb{Z}$ and $A \in \Sigma^{\mathbb{Z}}$. As before, the array A is the tape configuration. The ***damage set*** is defined as

$$A.Damage = \{p : A[p] = Bad\}.$$

We say that there is a ***cell*** at position $p \in \mathbb{Z}$ if $A[p] \notin \{\text{Vac}, Bad\}$. In this case, we call the interval $p + [0, B)$ the ***body*** of this cell. Cells must be at distance $\geqslant B$ from each other, that is their bodies must not intersect. They are called ***adjacent*** if the distance is exactly B.

For all cells p, the value $A[p].cDir$ is required to point towards the head h, that is

$$A[p].cDir = \text{sign}(h - p).$$

Whenever $A[h] \neq Bad$ there must be a cell at position $\tilde{h}$ called the ***current cell*** with a body within $2.5B$ from h.

The array A is Vac everywhere but in finitely many positions.

Let

$$\mathrm{Configs}_M$$

denote the set of all possible configurations of a Turing machine M.

It is clear that all the above definition can be localized to define a configuration ***over a space interval*** I, where it is always understood that $h \in I$, that is I contains the head. ⌟

It is natural to name a sequence of configurations that is conceivable as a computation (faulty or not) of a Turing machine as "history". The histories that obey the transition function then could be called "trajectories". In what follows we will stretch and generalize this notion to encompass also some limited violations of the transition function.

In connection with any underlying Turing machine with a given starting configuration, we will denote by

$$Noise \subseteq \mathbb{Z} \times \mathbb{Z}_{\geqslant 0} \tag{6.2}$$

the set of space-time points (p, t), such that a fault occurs at time t when the head is at position p.

Definition 6.3 (History)**.** Let us be given a generalized Turing machine (6.1). Consider a sequence $\eta = (\eta(0), \eta(1), \dots)$ of configurations with $\eta(t) = (q(t), A(t), h(t), \tilde{h}(t))$, along with a noise set *Noise*. The ***switching times*** of this sequence are the times when one of the following can change: the state, the position $\tilde{h}(t)$ of the current cell, or the state of the current cell. The interval between two consecutive switching times is the ***dwell period***. The pair

$$(\eta, Noise)$$

will be called a ***history*** of machine M if the following conditions hold.

- We have $|h(t) - h(t')| \leq |t' - t|$.
- In two consecutive configurations, the content $A(t)[p]$ of the positions not in $h(t) + [-B, B)$, remains the same: $A(t+1)[n] = A(t)[n]$ for all $n \notin h(t) + [-B, B)$.

- At each noise-free switching time the head is on the new current cell, that is $\tilde{h}(t) = h(t)$. In particular, when at a switching time a current cell becomes Vac, the head must already be on another (current) cell.
- The length of any dwell period in which the head does not intersect noise or damage, is at most T.

Let

$$\text{Histories}_M$$

denote the set of all possible histories of M.

We say that a cell ***dies*** in a history if it becomes Vac.

It is clear that all the above definition can be ***localized*** to define a history ***over a space-time rectangle*** $I \times J$, where it is always understood that $h \in I$ for all times $t \in J$, that is I contains the head throughout the time interval considered.

⌟

The transition function δ of a generalized Turing machine imposes constraints on histories: those histories obeying the constraints will be called trajectories.

Definition 6.4. Suppose that the machine is in a state q, with current cell x, with cell content a, let $(a', q', j) = \delta(a, q)$. We say that the new content of cell x (including when it dies), the direction of the new position y from x, and the new state are ***directed by the transition function*** if the following holds. The new content of x is a', and the direction of y from x is j. In the new state q we have $q.NonAdj = \text{false}$ if $j = 0$ or the new current cell is adjacent, and true otherwise. ⌟

Definition 6.5 (Trajectory)**.** A history $(\eta, Noise)$ of a generalized Turing machine (6.1) with $\eta(t) = (q(t), A(t), h(t), \tilde{h}(t))$ is called a ***trajectory*** of M if the following conditions hold, during any noise-free time interval. Denote

$$(a, q, p) = (A(t)[\tilde{h}(t)], q(t), \tilde{h}(t)),$$
$$(a', q', j) = \delta(a, q).$$

Transition function. Suppose that there is a switch, and the shortest interval containing the body of the current cell x and the new cell y is at a distance at least $c_{\text{dam-dist-1}}B$ from damage, where

$$c_{\text{dam-dist-1}} = 0.5 \tag{6.3}$$

Then the new state, the cell content of x (including when it dies), and the direction of y from x are directed by the transition function. If y did not exist before then it is adjacent to x.

Further the length of the dwell period is bounded by T.

Spill. For constant

$$c_{\text{spill}} = 4, \tag{6.4}$$

during any noise-free time interval J, the damage may spill to at most a total distance of $c_{\text{spill}}B$ beyond its place at the beginning of J.

Attacking Damage. For constant

$$c_{\text{dam-dist-2}} = 3, \tag{6.5}$$

suppose that the body of the current cell x is disjoint from damage, and at least $c_{\text{dam-dist-2}}B$ removed from damage on the left. Suppose further that the transition function directs the head towards the right. Then whenever the head comes left of $x - c_{\text{dam-dist-2}}B$, the damage will recede by a distance at least B to the right from the body of x.

A similar property is required when "left" and "right" are interchanged.

Clearing Damage. For constant

$$c_{\text{dam-clear-t}} = 9, \quad c_{\text{dam-clear-s}} = c_{\text{dam-dist-2}} + 1, \tag{6.6}$$

suppose that at the beginning of a noise-free time interval J, the head is in a certain interval I of size B. Then before the head spends a cumulative time $c_{\text{dam-clear-t}}T$ in I during J (possibly while entering and exiting repeatedly), it will be found in a damage-free interval of size $c_{\text{dam-clear-s}}B$.

The above definition can also clearly be localized to some space-time rectangle just as the definition of history was. ⌟

6.2 Simulation

Until this moment, we used the term "simulation" informally, to denote a correspondence between configurations of two machines which remains preserved during the computation. In the formal definition, this correspondence will essentially be a code $\varphi = (\varphi_*, \varphi^*)$. As a matter of fact, the *decoding* part of the code is the more important. Indeed, we want to say that machine M_1 simulates

machine M_2 via simulation ϕ if whenever $(\eta, Noise)$ is a trajectory of M_1 then $(\eta^*, Noise^*$ defined by $\eta^*(\cdot, t) = \phi^*(\eta(\cdot, t))$ is a trajectory of M_1, where $Noise^*$ is computed by an appropriate

We will make, however, two refinements. First, it is not important to require this for arbitrary trajectories η of M_1: it is sufficient to consider those η for which the initial configuration $\eta(\cdot, 0)$ has been obtained by encoding, that is it has the form $\eta(\cdot, 0) = \varphi_*(\xi)$. So the encoding function comes in, after all.

But there is a more complex refinement. When a colony is in transition between encoding one simulated value to encoding another one, there may be times when the value represented by it before the transition is already not decodable from it, and the value after the transition is not yet decodable from it. For this reason, it is useful to define a notion of simulation decoding Φ^* that is a mapping between *histories*, not just configurations. We will not abuse this notion, but it will allow us a certain amount of looking back: the map Φ^* can use the configuration at the beginning of the work period for decoding.

We will also give another function to the mapping Φ^*. A history was defined above in Definition 6.3 as a pair $(\eta, Noise)$, so Φ^* also computes a new noise set: $\Phi^*(\eta, Noise) = (\eta^*, Noise^*)$. The meaning of this new noise set will be, just as in Definition 5.2 of sparsity, that $Noise^*$ will be obtained by deleting some small isolated parts of *Noise* that the error-correcting simulation can deal with.

Definition 6.6 (Simulation)**.** Let M_1, M_2 be two generalized Turing machines, and let

$$\varphi_* : \text{Configs}_{M_2} \to \text{Configs}_{M_1}$$

be a mapping from configurations of M_2 to those of M_1, such that it maps starting configurations into starting configurations. We will call such a map a ***configuration encoding***. Let

$$\Phi^* : \text{Histories}_{M_1} \to \text{Histories}_{M_2}$$

be a mapping. The pair (φ_*, Φ^*) is called a ***simulation*** (of M_2 by M_1) if for every trajectory $(\eta, Noise)$ with initial configuration $\eta(\cdot, 0) = \varphi_*(\xi)$, the history $(\eta^*, Noise^*) = \Phi^*(\eta, Noise)$ is a trajectory of machine M_2.

We say that M_1 ***simulates*** M_2 if there is a simulation (φ_*, Φ^*) of M_2 by M_1. ⌟

6.3 Hierarchical codes

Recall the notion of a code in Definition 1.2.

Definition 6.7 (Code on configurations)**.** Consider two generalized Turing machines M_1, M_2 with the corresponding state spaces, alphabets and transition functions, and an integer $Q \geqslant 1$. We require

$$B_2 = QB_1. \tag{6.7}$$

Assume that a block code

$$\psi_* : \Sigma_2 \times (\Gamma_2 \cup \{\emptyset\}) \to \Sigma_1^Q$$

is given, with an appropriate decoding function, ψ^*. With $(a, q) \in \Sigma_2 \times (\Gamma_2 \cup \{\emptyset\})$, symbol a is interpreted the content of some tape square. The value q is the state of M_2 provided the head is observing this square, and $\emptyset$ if it is not.

Block code (ψ_*, ψ^*) gives rise to a ***code on configurations***, that is a pair of functions

$$\varphi_* : \mathrm{Configs}_{M_2} \to \mathrm{Configs}_{M_1}, \quad \varphi^* : \mathrm{Configs}_{M_1} \to \mathrm{Configs}_{M_2}$$

that encodes configurations of M_2 into configurations of M_1.

Let ξ be a configuration of M_2. We set $\varphi_*(\xi).\mathrm{pos} = \xi.\mathrm{pos}$, $\varphi_*.\mathrm{state}$ = the starting state of M_1,

$$\varphi_*(\xi).\mathrm{tape}[iB_2, \ldots, (i+1)B_2 - B_1] = \psi_*(\xi.\mathrm{tape}[i], s)$$

where $s = \xi.\mathrm{state}$ if $i = \xi.\mathrm{pos}$, and $\emptyset$ otherwise, with a slight *modification*: $\varphi_*(\xi).\mathrm{tape}[i].cDir$, is set to point towards the head $\xi.\mathrm{pos}$. Decoding is the inverse of this process (need not succeed on all possible configurations of M_1). ⌟

Not all configurations can be obtained by encoding. We distinguish those that can.

Definition 6.8 (Code configuration)**.** A configuration ξ is called a ***code configuration*** if it has the form $\xi = \varphi_*(\zeta)$. ⌟

Definition 6.9 (Hierarchical code)**.** For $k \geqslant 1$, let Σ_k be an alphabet, Γ_k be a set of states of a generalized Turing machine M_k. Let $Q_k > 0$ be an integer colony size, let φ_k be a code on configurations defined by a block code

$$\psi_k : \Sigma_{k+1} \times (\Gamma_{k+1} \cup \{\emptyset\}) \to \Sigma_k^{Q_k}$$

as in Definition 6.7. The sequence of triples $(\Sigma_k, \Gamma_k, \varphi_k)$, $(k \geqslant 1)$, is called a ***hierarchical code***. For the given hierarchical code, the configuration ξ^1 of M_1 is called a ***hierarchical code configuration*** if a sequence of configurations $\xi^2, \xi^3, \ldots$ of $M_2, M_3, \ldots$ exists with

$$\xi^k = \varphi_{*k}(\xi^{k+1})$$

for all k. (Of course, then whole sequence is determined by ξ^1.) ⌟

Let us give a name to the object that we want to construct.

Definition 6.10 (A tower)**.** Let $M_1, M_2, \ldots$, be a sequence of generalized Turing machines, let $\varphi_1, \varphi_2, \ldots$ be a hierarchical code for this sequence, let ξ^1 be a hierarchical code configuration for it, where ξ^k is an initial configuration of M_k for each k. Let further be a sequence of mappings $\Phi_1, \Phi_2, \ldots$ be given such that for each k, the pair (φ_{k*}, Φ_k^*), is a simulation of M_{k+1} by M_k. Such an object is called a ***tower***. ⌟

The main task of the work will be the definition of a tower, since the simulation property is highly nontrivial.

7 Simulation structure

In what follows we will describe the program of the reliable Turing machine (more precisely, a simulation of each M_{k+1} by M_k as defined above). Most of the time, we will just refer to M_k as M and to M_{k+1} as M^*. Along the way, we will try to give as much motivation as possible, including some definitions (like health) that will help in the proof later. There are two difficulties faced by our desire for a structured presentation.

- We cannot analyze unconditionally the error-correcting performance of a part of the program without seeing first the whole. Indeed, the noise can bring the machine into some state corresponding to an arbitrary part of the program.
- The program cannot refer explicitly to damage, since damage is not a particular property of cell symbols: rather, it is a certain unpredictability present on some parts of the tape. So the program will be written in a way as if it was not planned for damage, but its the later analysis will show that it prevails even in the presence of the allowed amount of damage, thanks to its various redundancies.

Ordinary simulation proceeds in the normal mode. The structure will be arranged in such a way that when the basic structure supporting this process is broken somewhere, then this will be recognizable by a one-step checking of a simple relation: whether the present state is ***coordinated*** with the currently observed cell symbol (see Definition 8.4). This condition is then checked in every step, and if it is found violated then the rule

$$\texttt{Alarm}$$

takes the state into the recovering mode. (The machine enters into the rebuilding mode if healing fails.) The crudest outline of the main rule of machine M is given in Rule 7.1, where the `Compute` and `Transfer` rules will be outlined below.

Rule 7.1: Main rule
if the mode is normal **then**
if not Coordinated **then** `Alarm`
else if $1 \le Sw <$TransferSw(1) **then** `Compute`
else if TransferSw(1) $\le Sw <$Last **then** `Transfer`
else if Last $\le Sw$ **then** move the head to the new base.

7.1 Sweep counter and direction

The global sweeping movement of the head will be controlled by the parametrized rule

$$\texttt{Swing}(a,b,u,v).$$

This rule makes the head swing between two extreme points a, b, while the counter Sw increases from value u to value v. The Sw value is incremented at the "turns" a, b (and is also recorded on the track cSw).

The sweep direction δ of the simulating head is derived from Sw, $Addr$ and the current value Dir in the following way. On arrival of the head to an endpoint (that is when $Dir \neq 0$ and $Addr \in \{a, b\}$), the values Sw and cSw are incremented and Dir is set to 0. In all other cases, the sweep direction is determined using the formula

$$\mathrm{dir}(s) = (-1)^{s+1} \tag{7.1}$$

as follows:

$$\delta = \begin{cases} 0 & \text{if } Addr \in \{a, b\} \text{ and } Dir \neq 0, \\ \mathrm{dir}(Sw) & \text{otherwise.} \end{cases} \tag{7.2}$$

As mentioned, each sweep will be broken up into zigzags to allow the detection of premature turn-back. At each non-zigging step, $Addr \leftarrow Addr + \delta$.

As an example of rules, we present the the zigging rule. Certain bursts may turn back the head prematurely, causing a skip in the simulation. We want to prevent this, since we would like the size of the structural repairs to be comparable to the burst size. The zigging rule uses the following parameter: ⟪Peter: I denote by Z what was denoted $Z - 4\beta$ before, and the 4β became 1⟫.

Definition 7.1. Let $Z = 40\beta$. ⌟

Rule 7.2, itself uses the rule `Move`(d), which we will defined later. This rule changes nothing on the tape: all its control information is in the state. The rule repeats the following cycle: first it moves forward one step, moving ahead the ***front***, and performing all necessary operations of the computation. Then it moves backward and forward by Z steps, not changing anything on the tape (but as we will see, some consistencies will be checked). The field *ZigDepth* of the state measures the distance from the front during the switchback.

Rule 7.2: `Zigzag`(d)

```
// d ∈ {−1, 1} is the direction of progress.
if ZigDir = −1 then
    if ZigDepth = Z then ZigDir ← 1
    else
        ZigDepth++
        Move(−d)                    // Move only when not at the bottom.
else if ZigDir = 1 then
    if ZigDepth = 0 then ZigDir ← −1
    else ZigDepth−−
    Move(d)                         // Move once even at the top.
```

7.2 Computation phase

As shown in Rule 7.1 describing a top-down view of the simulation, the first phase of the simulation computes new values for state of the simulated machine M^* represented on track *cState*, the direction of the move of the head of M^* (represented in the *cDrift* field of each cell of the colony of M), and the simulated cell state of M^* represented on the track *cInfo*. During this rule, the head sweeps the base colony.

The rule `Compute` essentially repeats five times the following ***stages***: decoding, applying the transition, encoding, checking for destruction.

Due to the possibility of encountering a much larger burst of faults than this rule can handle, some extra rules will then be applied that we will specify later: `UsefulComp`.

In more detail:

1. For every $j = 1, \ldots, 5$, if $Addr \in \{0, \ldots, Q-1\}$ do

   a) Calling by g the string found on the *cState* track of the base colony between addresses *PadLen* and $Q - PadLen$, decode it into string $\tilde{g} = v^*(g)$ (this should be the current state of the simulated machine), and store it on some auxiliary track in the base colony. Do this by simulating the universal machine on the *cProg* track, $\tilde{g} = \mathrm{Univ}(p_{\mathrm{decode}}, g)$.

   Proceed similarly with the string a found on the *cInfo* track of the base colony, between addresses *PadLen* and $Q - PadLen$, to get $\tilde{a} = v^*(a)$ (this should be the observed tape symbol of the simulated machine).

   b) Compute the value $(a', g', d) = \delta^*(\tilde{a}, \tilde{g})$. Since the neither the table nor any program of the transition function δ^* is written explicitly anywhere, this "self-simulation" step needs some elaboration, see Section 7.3.

   c) Write the encoded new state $v_*(g')$ onto the $cHold[j].State$ track of the base colony between positions *PadLenB* and $Q - PadLenB$. Similarly, write the encoded new observed cell content $v_*(a')$ onto the $cHold[j].Info$ track. Write d into the $cHold[j].Drift$ field of *each cell* of the base colony.

   Special action needs to be taken in case the new state g' is a vacant one, that is $g'.Kind^* = \mathrm{Vac}^*$. In this case, write 1 onto the $cHold[j].Doomed$ track (else 0).

2. Repeat the following twice (hoping that at least one repetition will be burst-free):

   > Sweeping through the base colony, at each cell compute the majority of $cHold[j].Info$, $j = 1, \ldots, 5$, and write into the field *cInfo*. Proceed similarly, and simultaneously, with *State* and *Drift*.

3. Call the rule `UsefulComp` that we will specify later.

It can be arranged—and we assume so—that the total number of sweeps of this phase, and thus the starting sweep number of the next phase, depends only on Q.

7.3 Self-simulation

In describing the rule of the computation phase, in the step 1b of Section 7.2, we said that machine M writes the code p^* of M^* onto the *cProg* track, without saying how this is done. Here we give the details.

New primitives

We will make use of a special track

$$cWork$$

of the cells and the special field

$$Index$$

of the state of machine M that can store a certain address of a colony.

Recall from Section 4.2 that the program of our machine is a list of nested "**if** *condition* **then** *instruction* **else** *instruction*" statements. As such, it can be represented as a binary string

$$R.$$

If one writes out all details of the construction of the present paper, this string R becomes completely explicit, an absolute constant. But in the reasoning below, we treat it as a parameter.

There is a couple of ***extra primitives*** in the rules. First, they have access to the parameter k of machine $M = M_k$, to define the transition function

$$\delta_{R,k}(a, q).$$

The other, more important, new primitive is a special instruction

`WriteRulesBit`

in the rules. When called, this instruction makes the assignment $cWork \leftarrow R(Index)$. This is the key to self-simulation: *the program has access to its own bits.* If $Index = i$ then it writes $R(i)$ onto the current position of the *cWork* track.

Simulating the rules

By convention, in our fixed flexible universal machine Univ, program p and input x produce an output $\mathrm{Univ}(p, x)$. Since the structure of all rules is very simple, they can be read and interpreted by Univ in reasonable time:

Theorem 2. *There is a constant string called Interpr with the property that for all positive integers k, string R that is a sequence of rules, and bit strings $a \in \Sigma_k$, $q \in \Gamma_k$:*

$$\mathrm{Univ}(Interpr, R, 0^k, a, q) = \delta_{R,k}(a, q).$$

The computation on Univ *takes time* $O(|R| \cdot (|a| + |q|))$.

The proof parses and implements the rules in the string R; each of these rules checks and writes a constant number of fields.

Implementing the `WriteRulesBit` instruction is straightforward: Machine Univ determines the number i represented by the simulated *Index* field, looks up $R(i)$ in R, and writes it into the simulated *cWork* field.

Note that there is no circularity in these definitions:

- The instruction `WriteRulesBit` is written *literally* in R in the appropriate place, as "`WriteRulesBit`". The string R is *not part* of the rules (that is of itself).
- On the other hand, the computation in $\mathrm{Univ}(Interpr, R, 0^k, a, q)$ has *explicit* access to the string R as one of the inputs.

Let us show the computation step invoking the "self-simulation" in detail. In the earlier outline, step 1b of Section 7.2, said to compute $\delta^*(\tilde{a},\tilde{g})$ (for the present discussion, we will just consider computing $\delta^*(a,q) = \delta_{k+1}(a,q)$), where $\delta = \delta_k$, and it is assumed that a and q are available on two appropriate auxiliary tracks. We give more detail now of how to implement this step:

1. Onto the *cWork* track, write the string R. To do this, for *Index* running from 1 to $|R|$, execute the instruction `WriteRulesBit` and move right. Now, on the *cWork* track, replace it with $\langle Interpr, 0^{k+1}, R, a, q\rangle$. Here, string *Interpr* is a constant, so it is just hardwired. String R already has been made available. String 0^{k+1} can be written since the parameter k is available. Strings a, q are available on the tracks where they were stored.

2. Simulate the universal automaton Univ on track *cWork*: it computes $\delta_{R,k+1}(a,q) = \mathrm{Univ}(Interpr, R, 0^{k+1}, a, q)$ as needed.

   This achieves the forced simulation. Note what we achieved:

   - On level 1, the transition function $\delta_{R,1}(a,q)$ is defined completely when the rule string R is given. It has the forced simulation property by definition, and string R is *"hard-wired"* into it in the following way. If $(a',q',d) = \delta_{R,1}(a,q)$, then

     $$a'.cWork = R(q.Index)$$

     whenever $q.Index$ represents a number between 1 and $|R|$. and the values $q.Sw$, $q.Addr$ satisfy the conditions under which the instruction `WriteRulesBit` is called in the rules (written in R).

   - The forced simulation property of the *simulated* transition function $\delta_{R,k+1}(\cdot,\cdot)$ is achieved by the above defined computation step—which *relies on* the forced simulation property of $\delta_{R,k}(\cdot,\cdot)$.

Remark 7.2. This construction resembles the proof of Kleene's fixed-point theorem. ⌟

7.4 Transfer phase

If the simulated machine head moves left or right, then another phase will be added to the simulation: transferring the simulated state information to the neighbor colony, and to move there. Let us call the direction of the transfer the ***drift***.

During this phase, the range of the head includes the base colony and the neighbor colony determined by the drift, including a possible bridge between them.

The sweep in which we start transferring in direction δ is called TransferSw(δ), the ***transfer sweep***. We have TransferSw(-1) = TransferSw(1) + 1.

General structure of the phase

We will make use of some extra rules that we will specify in more detail later, but whose role is spelled out here.

The phase consists of the following actions.

1. Spread the value δ found in the cells of the *cDrift* track (they should all be the same) onto the neighbor colony in direction δ.

   There are some details to handle in case the neighbor colony is not adjacent: see Section 7.4.

2. For $i = 1, 2, 3$:

   Copy the content of *cState* track of the base colony to the $cHold[i].State$ track of the neighbor colony.

3. Repeat the following twice:

   Assign the field majority: $cState \leftarrow \text{maj}(cHold[1 \ldots 3].State)$ in all cells of the neighbor colony.

4. If $Drift = 1$, then move right to the left end cell of the neighbor colony (else you are already there).

5. In the last sweep (possibly identical with the move step above), in the base colony, if the majority of $cHold[j].Doomed$, $j = 1, \ldots, 5$, is 1 then turn the scanned cell into a stem cell: in other words, carry out the destruction.

Transfer to a non-adjacent colony

Let us address the situation when the neighbor colony is not adjacent.

Definition 7.3 (Adjacency of cells)**.** Cells a and b are ***adjacent*** if $|a - b| = B$. Otherwise, if $B < |a - b| < 2B$, then a and b are two ***non-adjacent neighbor cells***. For the sake of the present discussion, a ***colony*** is a sequence of Q adjacent cells whose *cAddr* value runs from 0 to $Q-1$. It may be extended by a bridge of length up to $Q - 1$, in the direction of the drift.

If the bodies of two cells are not adjacent, but are at a distance $<B$ then the space between them is called a ***small gap***. We also call a small gap such a space between the bodies of two colonies. On the other hand, if the distance of the bodies of two colonies is $>B$ but $<QB$ then the space between them is called a ***large gap***. ⌟

In the transfer phase, in order to know in a robust local way where the head is, the *cKind* field of the cells visited will be set as follows. The base colony has cells of kind Member to begin with. The kind of the cells of the neighbor colony, the target of the transfer, will be set as Target for the duration of the transfer. However, in the first transfer sweep, if there was a gap between the base and the target, then cells between them will be created or adapted to form a bridge, that extends the base colony, also extending its addresses. A bridge can override an opposite old bridge ("old" meaning that its *Sw* is maximal) or move into an empty area, or kill opposite bridge cells or stem cells while it extends. If while forming a bridge, another colony is encountered before the bridge grows to length QB), then this new colony's cells will get the kind Target, and add it all to the workspace. (There can be a gap of size $<B$ between the bridge and the neighbor colony.) Otherwise the bridge itself becomes this neighbor colony, and its cell kinds are turned to Target on the way back.

Recall that the *NonAdj* field of the state determines if the current cell is not adjacent to the cell where the head came from. After the transfer stage, we update the $NonAdj^*$ field encoded in the *cState* track of the neighbor colony: it becomes 1 if either there is a nonempty bridge, or there is a gap (found with the help of the *NonAdj* field) between the base colony and the neighbor colony.

This is done in part 2 of Section 7.4 again three times, storing candidate values into $cHold[j].NonAdj$ and repeated with everything else.

7.5 Notes on zigging

The zigging rule was introduced in Section 7.1. We add now the following refinements to it.

When the head steps outside of the workspace and does not find a cell, then it creates a stem cell whose *cDrift* is set to $-Dir$, where *Dir* is the field of the state storing the direction of the last step (see 4.2). Also its address *cAddr* continues the adresses of the workspace, modulo Q. While zigging outside the workspace in normal mode, we allow one small gap, next to the workspace (other situations cause alarm).

8 Integrity of the structure

The main part of the simulation uses an error-correcting code to protect information stored in *Info* and *State* fields. However, faults can ruin the simulation structure and disrupt the simulation itself. The error-correcting capabilities of the code used to store the information on the *Info* and *State* tracks, will preserve the content of these tracks as long as coding-decoding process implemented in the simulation is carried out. The structural integrity of a configuration is maintained with the help of a small number of fields. Below we outline the necessary relations among them allowing the identification and correction of local damage.

8.1 Healthy configuration

A configuration with local structural integrity will be called healthy. No cell in such a configuration should have marks of a healing or rebuild procedure. Larger bursts introduce new, non-local anomalies: these we will "legalize", since they might encode some conceivable configuration of the machine we simulate. Cells of a healthy configuration are grouped into gapless colonies, and a few transitional segments described below. The big picture is this:

- There is a base colony, possibly extended by a bridge in the direction of the drift. Possibly, in the direction of the drift, the neighbor colony of the base is a ***target***, meaning that its cells are marked as such.
- Non-base colonies are called ***outer colonies***. If an outer colony is not adjacent to its neighbor colony closer to the base, then it is extended by a bridge that covers this gap.

  A partial exception is the colony C closest to the base colony in the direction of the drift. If there is a gap between it and the base colony then initially it is covered by a bridge extending C, but in the first transfer sweep this bridge will be overridden by the bridge extending the base colony.

⟪Peter: Pictures!⟫

Definition 8.1 (Segments)**.** An ***(homogenous) segment*** is a sequence of adjacent neighbor cells, with addresses growing to the right, with the same value of *Sw* and *cDrift*, or a sequence of neighboring (not necessarily adjacent) stem cells (this will be called a ***desert***). We require such a segment to consist of cells of the same kind or only of colony cells and bridge cells. Its ***left end*** is the left edge of its first cell, and its ***right end*** is the right edge of its last cell.

It is a ***colony*** if the addresses grow from 0 to $Q-1$, possibly continued by a bridge on one side. It is a ***target*** if it consists of target cells. A target is never extended by a bridge.

A boundary is called ***rigid*** if its address is the end address of a colony in the same direction.

A ***boundary pair*** is a right boundary followed by a left boundary that is at distance $<B$ from it. It is a ***hole*** if the distance is greater than 0. It is ***rigid*** if at least one of its elements is. ⌟

In a healthy configuration, cells fall into certain categories. Outer cells are member cells in colonies other than the ones that are currently being manipulated.

Definition 8.2 (Outer cells)**.** Recall the definition of the sweep value $\mathrm{Last}(\delta)$ from (4.1). For $\delta \in \{-1,1\}$, if a cell is stem or $cDrift = \delta$, $cSw = \mathrm{Last}(\delta)$ then it will be called a ***right outer cell*** if $\delta = -1$, $-Q<cAddr<Q$, and a ***left outer cell*** if $\delta = 1$, $0 \le cAddr<2Q-1$. ⌟

According to this definition, a stem cell is both left and right outer cell.

Recall the definition of the ***transfer sweep*** $\mathrm{TransferSw}(\delta)$ in Section 7.4, if $\delta \neq 0$. (There is no transfer sweep if $\delta = 0$.)

Definition 8.3 (Healthy configuration)**.** The ***health*** of a configuration ξ of a generalized Turing machine M will be defined over a certain interval A. So it depends only on $\xi_{\mathrm{tape}}(A)$ further on whether ξ_{pos} is in A and if it is, where. But we will mention the interval A explicitly only where it is necessary. In particular, some of the structures described below may fall partly or fully outside A.

We require the mode is normal, and that the following conditions hold. Let $\delta = Drift$.

Normality. No cell in I is marked for healing or rebuilding: that is, for every $x \in I$, $cHeal.Sw = 0$ and $cRebd.Sw = 0$ (see the definition of marking after (4.2)).

Segments. The ***base*** is defined by counting back from $Addr$ such that passing from a target cell to a member, we set $Addr$ to $cAddr$. The cell closer to the base where the $Addr$ is adjusted during this count-back is called the ***address jump***.

All cells can be grouped into full extended colonies, with possibly some stem cells between these. In more detail:

- A segment of the ***extended base colony*** consisting of member cells.
- ***Extended outer colonies***, consisting of outer member cells.
- A segment of a possible target, defined by the value of *cSw* in its cells.
- Desert filling out the gaps between these other parts.

The colony starting at the base is the base colony, and it consists of member cells. To describe the other segments, we consider several cases.

- If $\delta = 0$ or $Sw<\text{TransferSw}(\delta)$, then there are no bridge or target cells.
- If $\text{TransferSw}(\delta) + 1<Sw<$ (the last two sweeps for δ), then there is a target colony in the direction δ. If its distance from the base colony is $\geqslant B$ then the bas colony is extended by a bridge filling the gap.
- If $Sw = \text{TransferSw}(\delta)$ then the above described situation is in the making, as a bridge segment is being built up in direction δ, or after that, a target segment is being built in direction δ, converting member cells into target cells.
- If $Sw = \text{TransferSw}(\delta)+1$ and there is still not a complete target colony, then the whole bridge is being converted into a target, as the head is traveling in direction $-\delta$.
- In the last sweep, the target cells are being converted into member cells.

These are the only possible segments to be seen in a healthy area.

The front. The farthest position $\text{front}(\xi)$ to which the head has advanced before starting a new backwards zig is called the ***front***: it can be computed from ξ.pos and *ZigDepth*, but can also be reconstructed from the *cSw* track.

Workspace. Suppose that $\text{front}(\xi)$ is inside the extended base colony or the target. The ***workspace*** is an interval of non-outer cells, such that:

- For $Sw<\text{TransferSw}(1)$, it is equal to the base colony.
- In case of $Sw = \text{TransferSw}(1)$, it is the smallest interval including the base colony and the cell neighboring to $\text{front}(\xi)$ on the side of the base colony.
- In case of $Sw = \text{TransferSw}(-1)$, it is the smallest interval including the base colony, the right neighbor colony, and the cell neighboring to $\text{front}(\xi)$ on the side of the base colony.
- If $\text{TransferSw}(-1)<Sw<\text{Last}(\delta)$, then it is equal to the union of the extended base colony and the target.
- When $Sw = \text{Last}(\delta)$, it is the smallest interval including the future base colony and $\text{front}(\xi)$.

《Peter: I hope to avoid yarding.》

Drift. If $Sw \geqslant \mathrm{TransferSw}(\delta)$ or $Sw = 1$ then *cDrift* is constant on the workspace.

A tape configuration is called ***healthy*** on an interval A when there is a head position (possibly outside A) and a state that turns it into a healthy configuration. ⌟

Note that health only depends on the fields in *Core* and the zigging field Z in the state, further the *cCore* field and the lack of marks in the cell content.

A violation of the health requirements can sometimes be noted immediately:

Definition 8.4 (Coordination)**.** The state of the machine is ***coordinated*** with the current cell if it possible for them to be together in a healthy configuration. ⌟

Recall that in Rule 7.1 it is required in normal mode, if lack of coordination is discovered then alarm is called. The following lemmas show the power of coordination: how local consistency checking will help achieve longer-range consistency.

Lemma 8.5. *In a healthy configuration, each* $Core = (Addr, Sw, Drift, Kind)$ *value along with* Z *determines uniquely the cCore value of the cell it is coordinated with, with the following exceptions.*

- *During the first transferring sweep, while creating a bridge between the base colony and the target colony, the front can be a stem cell or the first cell of an outer colony.*
- *Every jump backward from the target colony can end up on the last cell of a bridge (whose address is not recorded in the state) or the last cell of the base colony.*

Proof. To compute the values in question, calculate Z steps backwards from the front, referring to the properties listed above. 《Peter: Elaborate!》 □

Lemma 8.6. *Suppose that during a noise-free time interval* J*, the head passes at least once over the two endcells of a damage-free space interval* I *(possibly passing over some parts of* I *several times). Then at the end of time interval* J*, the configuration over* I *is healthy.*

Proof. This follows from the above lemma, after examining some simple possibilities. 《Peter: Elaborate!》 □

Lemma 8.7 (Health extension)**.** *Let ξ be a* tape *configuration that is healthy on intervals A_1, A_2 where $A_1 \cap A_2$, contains a whole cell body of ξ. Then ξ is also healthy on $A_1 \cup A_2$.*

Proof. The statement follows easily from the definitions. □

In a healthy configuration, the possibility of finding non-adjacent neighbor cells is limited.

Lemma 8.8. *An interval of size $<Q$ over which the configuration ξ is healthy contains at most two maximal sequences of adjacent non-stem neighbor cells.*

Proof. Indeed, by definition a healthy configuration consists of full extended colonies, with possibly stem cells between them. An interval of size $<Q$ contains sequences of adjacent cells from at most two such extended colonies. □

Let us classify the boundary pairs possible in a healthy configuration. Rigid pairs:

(r1) Between an outer extended colony or a desert, and a colony closer to the base.

(r2) Between the extended base colony and the target or an outer colony.

Non-rigid pairs are the place of the front, but it is worth distinguishing some kinds:

(nr1) Between sweep values differing by 1, between a new bridge and the target that it is possibly being converted into, between a target in direction δ and the remaining segment of member cells, in direction $-\delta$, or between the target and the member cells replacing it in the last sweep.

(nr2) Boundary of a new bridge in direction δ not within distance B of a rigid boundary of a colony or target in direction $-\delta$. On the other side is either desert or the boundary of an old bridge.

The following is worth noting.

Lemma 8.9. *In a healthy configuration, any interval of size $<Q$ contains at most 3 boundary pairs, only one of which can be a hole.*

Proof. Any interval of size $<Q$ contains at most one rigid left boundary and one rigid right boundary. Of the boundary pairs containing the other types, only one can be present: indeed, they all coincide with the location of the front. □

Definition 8.10 (Super healthy)**.** A configuration is ***super healthy*** if in addition to the requirements of health, in each colony, whenever the head is not in the last sweep, the *Info* and *State* tracks contain valid codewords as defined in Section 5.2.

A configuration ξ defined on an interval I is ***(super) healthy*** on I if it can be extended to a (super) healthy configuration. ⌟

8.2 Annotation

Configurations that have been affected by noise in a limited way only, can be "annotated", adding some information showing some of the affected parts.

We will make use of a new parameter E which approximately measures the work area needed for eliminating the damage caused by a couple of bursts. Let:

$$E = 30Z, \tag{8.1}$$

$$\text{PenetLen} = E + Z. \tag{8.2}$$

The definition below makes use of the constant

$$c_{\text{island}} = 3. \text{ ⟪Peter: ?⟫}$$

Definition 8.11 (Annotated configuration)**.** An ***annotated configuration*** of machine M whose cell body size is B, is a tuple

$$\mathcal{A} = (\xi, \chi, \mathcal{I}, D),$$

with the following meaning.

ξ is a configuration.

χ is a healthy configuration,

$\mathcal{I}$ is a set of intervals called ***islands***, each of size $\leq c_{\text{island}}\beta B$. They may contain an interval of damage, of size $\leq \beta B$.

H is an interval containing the head called the ***healing area***[1]. This is where the structure is currently being restored. It contains any island containing the head. It has size $<5EB$.

We can obtain χ from ξ by removing damage from the islands, filling them with cells, and then by

- changing the state,

[1] The corresponding set in [1] called the "distress area".

- changing the *cKind*, *cCore* and *cHeal.Core* tracks in the islands and possibly additional $\leq Z$ cells within D, where *cHeal.Core* is a set of fields used in the healing procedure;
- changing the *cHeal.Core* track, *cKind*, and the head position inside D.

The ***current colony*** of $\mathcal{A}$ is the base colony of χ.

We say that an interval W is the ***workspace*** of the annotated configuration $\mathcal{A}$ if it is the workspace of χ.

The following additional properties are required:

(a) At most 1 island intersects the workspace.

There are at most 2 islands in each extended colony that do not intersect the workspace. If there is more than one, then one of them is within a distance $\mathrm{PenetLen} \cdot B$ from the boundary of the neighbor colony towards the base colony.

(b) If D is empty then the mode is normal.

We say that a cell is ***free*** in an annotated configuration when it is not in any island or D. The head is ***free*** when D is empty. ⌟

Here are some notes on the requirements above. First, on the requirements concerning islands. Normally, if an island is created by a burst, it will be eliminated in the next sweep at the latest, before other islands could be created. If an island is created in the last sweep before leaving a colony, then it may be revisited only much later; but then it will be eliminated in the first transfer sweep, before the workspace extends over it. This accounts for one island per extended colony outside the workspace. But during the normal operation of the workspace, while the head is at the edge or slightly outside due to zigging or healing, a burst can create another island. The careful construction of the healing process (making the boundary address of the healing interval $[a, b)$ a multiple of E) will guarantee that only one such additional island can remain in extended colonies outside the workspace.

Definition 8.12. A tuple $(\xi, \chi, \mathcal{I}, D, \mathcal{S})$, is a ***super annotated configuration*** if $(\xi, \chi, \mathcal{I}, D)$ is an annotated configuration, with the following additional properties. $\mathcal{S} \supset \mathcal{I}$ is a set of intervals of cells called ***stains***. In the base colony, either all stains but one are within a distance $E + \beta B$ to the left colony boundary, or all but one are within a distance $E + \beta B$ to the right colony boundary. The one exception is called the ***internal stain***. In all other colonies, all stains but one are within distance $E + \beta B$ of the boundary towards the base colony. The configuration ξ is super healthy (see the definition of health), and we obtain it by

the above operations and, in addition, by changing the *cInfo* and *cState* track in the stains.

⌟

Configurations that have an annotated configuration deserve a special name.

Definition 8.13 (Admissible configuration)**.** A configuration ξ is a ***(super) admissible*** if there is a (super) annotated configuration $(\xi, \dots)$. In this case, we say that χ is a (super) healthy configuration ***satisfying*** ξ. Any change to an admissible configuration is called ***admissible***, if the resulting configuration is also admissible. ⌟

Intuitively, the definition says that a configuration is admissible on an interval, if there are not "too many" islands in that interval, and that by local changes, we can obtain a corresponding healthy configuration on the same interval. The healing procedure will maintain this property under conditions of sparse bursts.

In Lemma 8.7 (Health extension) we have seen that a tape configuration that is healthy on overlapping intervals is healthy on the union. We would like to establish something similar for admissibility. Recall Definition 8.1 of segments.

Definition 8.14 (Substantial segments)**.** Let $\xi(A)$ be a tape configuration admissible over an interval A. We will call a homogenous segment of $\xi(A)$ ***substantial*** if it has size at least $4c_{\text{island}}\beta B$. The area between two neighboring maximal substantial segments or between an end of A and the closest substantial segment will be called ***ambiguous***. It is ***terminal*** if it contains an end of A. Let

$$\Delta = 22c_{\text{island}}\beta, \quad \Delta' = \Delta + 3c_{\text{island}}\beta.$$

⌟

The choice $3\beta B$ above is made to make it larger than the total length of the 2 possible islands that can interrupt any homogenous segment of the healthy configuration.

Lemma 8.15. *In an admissible tape configuration ξ, the size of each ambiguous area is at most ΔB.*

Proof. Let χ be any healthy configuration satisfying ξ in an appropriate annotation. There are at most 3 boundary pairs in χ at a total size of $3B$, further 3 islands in the annotation, of size $\leq c_{\text{island}}\beta B$, with 4 non-substantial segments of sizes $<4c_{\text{island}}\beta B$ between them: this adds up to

$$<(3 + 3c_{\text{island}}\beta + 4 \cdot 4c_{\text{island}}\beta)B<22c_{\text{island}}\beta B = \Delta B.$$

□

Just as we distinguished super healthy configurations, we will also distinguish super annotated ones. *Stains* will account for islands that have been healed, but whose information is lost for the purposes of simulation.

Definition 8.16 (Local configuration, replacement)**.** A ***local configuration on*** a (finite or infinite) interval I is given by values assigned to the cells of I, along with the following information: whether the head is to the left of, to the right of or inside I, and if it is inside, on which cell, and what is the state.

If I' is a subinterval of I, then a local configuration ξ on I clearly gives rise to a local configuration $\xi(I')$ on I' as well, called its ***subconfiguration***: If the head of ξ was in I and it was for example to the left of I', then now $\xi(I')$ just says that it is to the left, without specifying position and state.

Let ξ be a configuration and $\zeta(I)$ a local configuration that contains the head if and only if $\xi(I)$ contains the head. Then the configuration $\xi|\zeta(I)$ is obtained by replacing ξ with ζ over the interval I, further if ξ contains the head then also replacing ξ.pos with ζ.pos and ξ.state with ζ.state. ⌟

9 Stitching

In this section we show that an admissible configuration of machine M can be locally corrected. Moreover, in case the configuration is damage-free then this correction can be carried out by the machine M itself. We will deal with the elimination of damage later.

Recall the notion of substantial and ambiguous segments in Definition 8.14. We introduce an operation called ***stitching*** below that handles ambiguous areas between substantial segments. First we handle the simplest case, when the two segments should be merged into one.

Definition 9.1 (Mergeable segments)**.** Two substantial segments $[a_1, b_1)$ and $[a_2, b_2)$ will be called ***mergeable*** if they are connected to each other with at most $\Delta\beta$ other neighbor cells, and if $0<cAddr'(a_2) - cAddr'(b_1 - 1) \leq \Delta\beta$. ⌟

It is easy to see that in an admissible configuration, two mergeable substantial segments can indeed be merged: one can erase the cells in the ambiguous area that cannot be added adjacently to the other ones, and replace them all with adjacent cells, having addresses growing from $cAddr(b_1)$ to $cAddr(a_2) - 1$. While doing this, *no cells outside the islands need be changed.*

Machine M can recognize a pair of mergeable substantial segments. If the configuration was admissible it can also merge them: if the merging fails then it can be concluded that the configuration is not admissible.

Definition 9.2 (Stitching)**.** In an admissible configuration ξ with a satisfying healthy configuration χ, consider two substantial segments I_1, I_2 with an ambiguous area J between them: so $I_1 \cup J \cup I_2$ is an interval. We define an operation called ***stitching*** on $\xi(J)$, performable by machine M, that turns the ξ over $I_1 \cup J \cup I_2$ into a healthy configuration.

If I_1 and I_2 are mergeable then we perform the above described merging operation.

If they are not mergeable then $\chi(J)$ contains one of the possible boundaries listed after Lemma 8.8. We will recreate such a boundary with the help of $\xi(I_1)$ and $\xi(I_2)$. In case that the boundary is rigid, its position will be what it was in χ, otherwise this is not necessary.

Consider a boundary of type (r1): for example a left extended outer colony, followed by the workspace on the right: base colony or target. Then extend the base colony into the ambiguous area to the left until the left colony end is reached (erasing and replacing any cells that are in the way). Now if there is a left outer extended colony, extend it to the right, until the left end of the base colony is met. If there is only the desert on the left then replace it with stem cells that extend the base colony to the left.

In case of a boundary of type (r2), again we extend the target or colony side first towards the separation, and then the bridge toward the new colony end.

Consider a boundary of type (nr1) covered by an ambiguous area: it must be between substantial segments that would be mergeable, except that they differ in their sweep values by 1 (and also possibly in the kind of cells, as colony or bridge cells are converted into target cells, or a target cells into member cells in the sweep of question). Then merge the two segments, set the sweep over the separation to be equal to the older sweep (with the necessary change in cell kinds), and place the head to the boundary of the two sweeps.

Consider a boundary of type (nr2), where a new bridge meets either an old bridge, a target, an old colony or desert. If it is meeting an old bridge or desert,

then extend the new bridge end into the ambiguous area until it reaches the other bridge or a colony. If it is meeting an old colony or a target then first extend the colony to its maximum, then extend the new bridge to meet it. If there is an old bridge to meet it, then extend the old bridge as much as possible. If it is meeting a desert then replace the desert with adjacent stem cells that extend the bridge. ⌟

10 The healing procedure

Structure repair will be split into two procedures. The first one, called ***healing***, performs only local repairs of the structure: for a given locally admissible configuration, it will attempt to compute a satisfying healthy configuration. If it fails—having encountered a configuration that is not admissible, or a new burst—then a so-called ***rebuilding*** procedure is called, which is designed to repair a larger interval. On a higher level of simulation, this corresponds to the implementation of the trajectory properties in which damage "magically" disappears. The healing procedure runs in $O(\beta^2)$ steps, whereas rebuilding needs $O(Q^2)$ steps. ⟪Peter: maybe even Q^3?⟫

The details of the section look as if we assumed that there is no noise or damage. The rules described here, however, will remove the damage under the appropriate conditions, and will also work under appropriately moderate noise.

The healing procedure designates an interval $[a, b)$ to which it applies the stitching operations defined above in Definition 9.2. This healing interval may be extended somewhat during the procedure.

10.1 Healing addresses

Since the number and position of some cells may change during healing, we will use the address field

$$cHeal.Addr$$

in a more complicated way than the main simulation. It will count the number of cells from the left end of the healing interval to the current cell if it is positive, and from the right end if it is negative. Starting from the head, *cHeal.Addr* decreases to 0 towards the left, and increases to 0 towards the right (its value under the head belongs to the left or the right segment depending on the direction of movement). Using two counts instead of one helps keeping track of

the position of important points, even though occasionally a new cell will be inserted at the head.

The updating will also use the fields $Heal.Addr_{-1}$ and $Heal.Addr_1$ of the state. When moving in direction j, field $Heal.Addr_{-j}$ remembers the last value of $cHeal.Addr$ written, and field $Heal.Addr_j$ the last one seen before writing. We also set $cHeal.Addr \leftarrow Heal.Addr_{-j} + j$. (Some modification will be needed for the case when one of the fields is 0.) ⟪Peter: !⟫

10.2 Healing stages

Whenever we say that a rule "checks" something, it is understood that if the check fails, the rule `Alarm` is called, restarting the healing from scratch. This will be judged insufficient in some cases: for example if the healing interval contains cells marked for the rebuilding procedure (see later), or the healing procedure fails in a way indicating that the underlying configuration is not admissible. This indicates a "bigger mess", and is followed by a call to the rule

`GiveUp`

that we will specify later. We will sometimes just say: then the machine ***gives up***, meaning that the `GiveUp` rule is called.

There is a field $cHeal.Dir$ used only to record the direction of movement in the special case when the head steps onto a vacant or stem cell.

The field $Heal.Sw$ measures the progress, just as Sw in the main program. There is a corresponding $cHeal.Sw$ field in the cells. According to the values of $Heal.Sw$, we distinguish ***stages***.

In the ***planning*** stages, wherever the head steps, it walks over marked cells. In the ***committing*** stages it may set or remove marks.

Each committing stage is β steps long. Even if more could be accomplished at that stage, a new planning stage is called which decides again what to do. This limits the possible effect of any burst interrupting healing. On the other hand, each planning stage is β *sweeps* long. The cause is not evident here, since it seems that a couple of sweeps should be sufficient to gather information. However, we want our healing procedure eventually to work even in the presence of *damage in the islands*. And damage can in principle kick back the healing sweeps repeatedly, while retreating only one cell at a time. So it may take β sweeps to pass through an island.

Suppose that `Alarm` is called at some position z. Then it sets $Mode \leftarrow$ Healing, $Stage \leftarrow$ Marking, $Heal.Sw \leftarrow 1$, and $Heal.Addr = 0$.

During planning stages, the following ***consistency check*** will be running all the time: (some modification may be needed when an address is 0): ⟪Peter: !⟫

- Check if $cHeal.Addr = Heal_j.Addr + j$, where $j = \pm 1$ is the direction of the sweep.
- Check if $cHeal.Sw = Heal.Sw - 1$.

Locating. Healing starts by marking and surveying an initial interval of Δ' cells to the left and Δ' cells to the right of z, in order to determine the direction δ in which a larger-scale marking should start. If no substantial (see Definition 8.14) homogenous segment is found then we give up; suppose that such a segment is found.

Suppose that at least one such substantial homogenous segment belongs to the workspace. If all such segments show the same sweep value σ then let $\delta = -\sigma$. If these segments show two sweep values ponting towards each other, then let δ be the direction of the older one.

Suppose that there is no workspace segment, but each substantial segment shows the same *Drift* value: then we call this value δ.

In all other cases we give up.

Marking. Next, the procedure marks out the initial healing area $[a, b)$. It marks every cell it passes, and alarms if any of the cells along the way that it expects to be marked is not. First, we mark $E/2$ cells in the direction δ from z, further E cells on the other side. (This way if a new burst drives the head away from an—even partially—marked area, it will be noticed that the front is left behind, allowing no orphaned marks to be left behind.)

Whenever the head steps on a stem cell or creates a new cell, *cDrift* is set to point to z (tomake sure that the head does not get lost in the desert). All this process is broken up into planning and commitment stages as outlined above.

Stitching. Next, in each planning stage the healing area is surveyed to gather information needed to carry out the stitching operations. A bounded number of substantial segments are found, (using some working tracks and working fields which we do not name), and it is determined what needs to be done to each of the boundaries between them. The first up to β steps of such a stitching operation are carried out in the next commitment stage, and then the process will be repeated. If it is concluded at any point that the configuration in question is not admissible then the `GiveUp` rule is called.

The planning and commitment stages are iterated enough times (just a large absolute constant) to make sure that everything inside R has been stitched.

In case that we started from an admissible configuration, the above operations created a sequence of substantial segments adjacent to each other, satisfying the requirements of a healthy configuration. After they were merged, then according to Lemma 8.8, they must form at most two ***domains*** consisting of adjacent cells, that extend to within ΔB of the boundaries a, b.

Extending. Let $a' \geqslant a$ be the left end of the leftmost one of the domains, and let $\alpha = cAddr(a')$. If a' is not the end of a bridge with $cDrift = -1$, then keep extending this domain to the left (adding an adjacent cell to the repair area and changing a' accordingly to $a' - B$), until $cAddr(a') \Leftrightarrow 0 \pmod{E}$. Any adjacent cell must be rewritten, any non-adjacent cell in the way must be killed. (This operation must also happen in planning and committing stages as described above.) Perform the corresponding operation also on the right end. Let $[a, b)$ denote the final domain $[a', b')$.

Trimming. Compute the position p of the front of the new configuration (or find out whether is outside $[a, b)$). Now we change the interval $[a, b)$ to achieve the property that if the front p is inside it then it is not near the boundary:

If $p - a < \Delta' B$ then increase a by $\Delta' B$. If $p - b \leq \Delta' B$ then decrease b by $\Delta' B$. While doing this, *the marks will not be erased.* (This is designed to cause a new alarm after the healing finishes, on the end where the trimming occurred.)

Mopping. Finally the marks are removed (again alternating planning and committing stages) in the following manner. If it the front is on the right of $[a, b)$, then they are removed starting from the left; if it is on the left then they are removed from the right. If it is inside then they are removed first from the end in direction δ to the front, (that is starting from behind the front), then from the other end to the front. This way, the healing area shrinks to the front while always containing the head.

11 Rebuilding

If healing fails, it calls the rebuilding procedure. This indicates that the colony structure is ruined in an interval of size larger than what can be handled by local healing. Just as with healing, we will be speaking here only about a situation with no damage—but the final analysis will take damage into account.

Since rebuilding makes changes on an interval of the length of several colonies, it is important not only that it is invoked when it is needed, but that it is not invoked otherwise! It will be invoked at the end of a failed healing, and it will be designed to fail immediately (to result in an alarm) if launched under normal conditions into any of its stages from just a burst. What assures this is that a rebuild starts from a marked area of a failed healing. The failure is determined in two planning stages, storing the result on two different tracks. Then a final sweep, checking that the two tracks identically show failure, turns the heal marks into rebuild marks, and thus the healing region into the germ of a rebuilding region. Rebuilding starts by extending the germ to the right. It will use zigging, expecting marked cells on its backward zig. It can find these in the first time only in the germ, so if rebuilding started from just a burst then alarm follows immediately.

The crude outline is this:

Marking. Mark a rebuilding area consisting of $4Q$ cells ⟪Peter: ?⟫ extending in both directions from the germ, similarly to marking a healing area.

Survey. Survey the rebuilding area, keeping track of addresses just as in the healing procedure, looking for candidate whole colonies (a few substantial segments with disjoint addresses fitting into a colony, with the allowed number of possible islands between them).

Stitching. For each candidate colony, attempt a stitching operation over each island. If one of the stitches in a candidate colony does not succeed, the candidate does not become a colony. Mark this result on a track, distinguishing accepted colony cells from the rest, which will be marked as ***stragglers***. Then repeat the whole operation and accept the end result only if the two tracks agree. Otherwise call alarm (which in due course may restart a rebuilding).

Creation. If there is no colony at all, create one, overriding any stragglers.

Initialize. Declare one of the colonies the base colony, in starting sweep. If there is only the base colony, grow other colonies from it in both directions. Put all other colonies into end sweep with drift towards the base colony, and grow bridges from them in the gaps between them towards the base colony. If any bridge gets Q cells, turn it into yet another colony. Throughout this, override any stragglers.

Mopping. Remove the rebuild marks similarly to the healing procedure.

How to defend against the effects of a burst during the rebuild procedure? Just as with healing, there is only limited defense. The major decision on which

cells belong to colonies is made twice, with the two results compared. Otherwise if the usual zigging (with simple consistency check on the rebuild addresses and sweeps) fails, alarm is called. This will most likely start another rebuilding, which now will operate without a burst. Traces (say, marked cells) from the first, interrupted rebuilding might remain, the analysis will deal with this possibility.

12 Resisting isolated bursts

In order to show what can be achieved by the healing procedure, we extend the notions of annotation and admissibility to histories.

12.1 Annotated history

Recall the definition of history and trajectory in Definitions 6.3 and 6.5.

Just as the notions of annotation and health for configurations were local to a certain interval, the definition of annotation, and so on, for histories will be local for a certain space-time rectangle. But just as there, we will not keep mentioning this all over the definition.

In a configuration in which the workspace is free, a cell is free when it does not belong to an island or to the healing area. We also said that the head is free if the healing area is empty.

Definition 12.1 (Annotated history)**.** An ***annotated history*** of a generalized Turing machine

$$M = (\Gamma, \Sigma, \delta, q_{\text{start}}, F, B, T)$$

is a sequence of annotated configurations over a certain time interval, if its sequence of underlying configurations is a (localized) trajectory satisfying some additional requirements given below.

We require that over the whole time interval, the head stays by at least a distance $3B$ inside from the ends of the space interval considered.

If the head is in a free cell, in normal mode, then the time (and the configuration) will be called ***distress-free***. A time that is not distress-free and was preceded by a distress-free time will be called a ***distress event***.

Consider a time interval $[t_1, t_1 + u)$ starting with a distress event and ending with the head becoming free again. It is called a ***relief event*** of ***duration*** u if the

only possible islands that remains from the healing area are due to some burst that occurred at a time intersecting $[t_1, t_1 + u)$. Moreover, if such islands exist, then the sweep direction from before the distress event is preserved, except when the island is outside the extended base colony—then it will be reversed.

The ***extent*** of a relief event is the maximum size interval covering the healing area during the distress.

The additional requirements for annotated history are:

(a) Islands (whose size do not exceed βB) are only created by bursts. The healing area starts out as an island.

(b) Each distress event is followed by a relief event within at most $O(\beta^2)$ steps of duration, and extent $<4EB$.

(c) If a distress-free configuration has $Sw \geqslant \text{TransferSw}(1)$, then the base colony contains no stains from earlier work periods.

⌟

In what follows we will show how any trajectory $(\eta, Noise)$ of a generalized Turing machine M that is annotated on a large interval I, remains annotated as long as no larger bursts than βB occur and the head does not leave I.

The main part of the proof is about obtaining relief after a distress event. Unlike in [1], in this work, islands may have their cell structure damaged. Hence, when the head is in an island for a certain time, we cannot talk immediately about the cells in it and their structure. However, since η is a trajectory, many passes over an island and a certain amount of time that the head spends in it, will guarantee that soon, we will be able to talk in terms of cells of M. This means that the relief consists of two stages: (1) damage-removal, and (2) correcting their structure. Note that this division is only possible for an observer. There is no "damage-detector" in the machine: we just rely on the properties of a trajectory introduced in Definition 6.5.

12.2 Escape

In the elimination of damage, we will use the following fact. Recall the constant $c_{\text{dam-clear-t}}$ from Definition 6.5 of trajectories.

Lemma 12.2 (Damage-free escape)**.** *There is a constant*

$$c_{\text{dam-free-esc}} \geqslant c_{\text{dam-clear-t}} \tag{12.1}$$

with the following property. Suppose that I is any interval of size $\leq nB$ for an integer n, that contains no damage. Then the head spends at most $c_{\text{dam-free-esc}}n^2$ noise-free steps inside without leaving it.

Proof for the case $n<Q$. The proof here begins a pattern of similar proofs that, unfortunately, involve a number of case distinctions.

1. Assume that at the beginning, the mode is normal. Then either escape or an alarm call happens within $O(\beta n)$ steps.

   Proof. The head begins or continues a sweeping motion that, with some zigging switchbacks, passes over an area at least the size of a colony. If it does not discover any incoordination then it escapes I within Zn steps, otherwise alarm will be called.

2. Suppose that at some time t while the head is inside I, alarm will be called. Then escape happens within $O(\beta n)$ steps.

   Proof. In the absence of damage or noise, no alarm will be called before the end of the healing procedure.

   If healing ends in a call to the rule `GiveUp` then, as outlined in Section 11, the rebuilding procedure is called. Since it started from failed healing, it starts from a sufficient base of marked cells to guarantee that no more alarm will be called before it ends. In its the first stage, the rebuilding procedure sweeps over an area of size greater than QB, assuring escape from interval I.

   Suppose therefore that healing ends successfully. In this case the configuration becomes healthy over the last interval $[a,b)$ of the healing procedure. The normal sweeping movement of the head is resumed. Lemma 8.6 implies that before any new alarm is called, all the cells that the head passes over can be added to this healthy configuration. If no more alarm is called then the head escapes as above. Suppose now that a new alarm is called (sometimes even before any sweeping movement could start).

   The new alarm is called outside the last healing interval $[a,b)$. It follows that the new healing interval $[a',b')$ differs from the old one by an area of size $\geqslant EB/2$. By a combination of Lemmas 8.6, and 8.7, at the time of success, the configuration is healthy on the smallest interval containing $[a,b)\cup[a',b')$.

   Continuing the above reasoning shows that in $O(\beta t)$ steps, ⟪Peter: Estimate!⟫, the healthy area extends to a size tB, eventually leading again to the escape

from I.
Now consider the possibilities for when the head is found in the interval I. The case when we start from normal mode, or if alarm is called within $O(\beta n)$ steps is treated above.
Suppose now that we start from the rebuilding mode. If alarm is not called within $O(\beta n)$ steps and does not turn into normal mode before escaping then the rebuild procedure will take it outside I just as seen above.
The case remains when we start in healing mode. If no new alarm is called within $O(\beta n)$ steps then the healing terminates, either in rebuilding mode or normal mode, and we are back to one of the cases already treated. □

We will prove the case $|I| \geqslant QB$ of the above lemma later, since it relies on a detailed analysis of rebuilding. The upper bound $O(n^2)$ is pessimistic, we may later improve it.

Let us apply Lemma 12.2 together with the basic trajectory properties to track the elimination of damage. First we need some definitions. Recall the trajectory properties in Definition 6.5, and the definition of the constants $c_{\text{dam-clear-s}}$ and $c_{\text{dam-dist-2}}$ there.

Definition 12.3 (Right-live)**.** Consider a damage-free interval of size $[a, b)$ of size $\geqslant c_{\text{dam-clear-s}}B$. Assume that an attacking event of the Attacking property takes place at point $x \leq b - B$ at the beginning time t of a noise-free time interval. So the body of cell x is contained in $[a, b)$ where a is at a distance $\geqslant c_{\text{dam-dist-2}}$ from the left edge of the body cell x, and b is to the right of the right edge y of x. From this time t on, we will call the interval $[a, y)$ a ***right-live*** interval until the first time when the head returns to the left of $x - c_{\text{dam-dist-2}}B$.

Left-live intervals are defined analogously. ⌟

While the interval $[a, y)$ is right-live, damage may spill over on the left of y as far as $c_{\text{spill}}B$, but this is temporary: after the return, according to the Spill and Attack properties, damage is cleared from the whole interval $[a, y + B)$.

Definition 12.4 (Cheese)**.** In a configuration, consider an interval I at a distance at least $(c_{\text{dam-dist-2}} + c_{\text{spill}})B$ from damage, and such that damage inside I is also removed by at last $c_{\text{spill}}B$ from the edges. Let us further have a sequence of subintervals $J_1, \ldots, J_k$ of I, where $|J_i| \geqslant (c_{\text{dam-dist-2}} + 1)B$ and $|J_i \cap J_{i+1}| \leq c_{\text{spill}}B$. For each J_i we require that if it is to the left of the head it is right-live, if it is to the right then it is left-live, and if it contains the head then it is damage-free. Such a structure $C = (I, J_1, \ldots, J_k)$ is called a ***cheese of the configuration*** with I called its ***bulk*** and the intervals J_i called its ***holes***. ⟪Peter: Picture?⟫

In a trajectory, over a certain time interval K, suppose that there is a cheese $C(t)$ at each time t. The whole sequence of cheeses is called a ***cheese of the trajectory*** if from one switching time to the next only one of the following changes can occur, within distance $c_{\text{dam-clear-s}}B$ of the head:

- One of the holes gets enlarged.
- A new hole is created.
- The spill at a live end of a hole changes within its allowed limits.

In each case, if the union of two holes becomes a single hole then we replace them with the new one.

We say that the head ***escapes*** the cheese if it is found in a damage-free interval of size $c_{\text{dam-clear-s}}B$ not contained in I. ⌟

Due to $c_{\text{dam-clear-s}} \geqslant 2c_{\text{spill}}$, the sum of hole sizes in a cheese is at most twice the size of its bulk.

Lemma 12.5 (Escape)**.** *Suppose that at some noise-free time t_0, the head is in the bulk of a cheese $C(t_0)$ of the configuration at time t_0, where the size of this bulk is $\leq nB$. Then the cheese can be extended to a cheese of the trajectory. If the head does not escape before some time $t \geqslant t_0 + 2i \cdot c_{\text{dam-free-esc}}n^2T$ such that $[t_0, t)$ is noise-free then by that time, the sum of the sizes of holes increases by at least iB. In particular, taking into account the remark after Definition 12.4, the head escapes before time t if this increase becomes greater than $2|C(t_0)|$.*

Proof. Since the damage inside the cheese begins at least $c_{\text{spill}}B$ removed from the edges of the bulk, the Spill property of a trajectory in Definition 6.5 implies that during the time interval considered, damage never spills beyond the bulk.

We will define a sequence of time intervals $t_0 < t_1 < t_2 \cdots < t$, where between t_i and t_{i+1} always some progress will be made.

Suppose that at time t_i the head is in a damage-free part of a hole. By Lemma 12.2, the head does not stay longer than

$$c_{\text{dam-free-esc}}n^2T$$

in the damage-free part of the hole. At some time t_{i+1} it either escapes, or steps onto damage. In the latter case the hole does not decrease, though it becomes right-live, possibly with some new spilled-over damage.

Suppose that at time t_i the head is over damage. Then it is either between (the damage-free parts of) two holes, say J_p and J_{p+1}, or between a hole and the

outside of the bulk. Assume that the head does not escape before an additional time

$$t' = t_i + c_{\text{dam-clear-t}} nT.$$

Suppose at some time $t_{i+1} \leq t'$ it steps into a hole, for example into J_{p+1} (possibly by merging to it another hole from the left). Since J_{p+1} was left-live, the Attack property in Definition 6.5 implies that the possible spilled-over damage on the left of J_{p+1} disappears—moreover J_{p+1} gets enlarged by at least the amount B.

Suppose that this does not happen. Let us partition the bulk into n subintervals of size B. Until time t' the head spends on average at least $c_{\text{dam-clear-t}} T$ over these subintervals, so there will be some subinterval over which it spent at least this much time. By the trajectory property about clearing damage, at some time t_{i+1} the head will be found in some damage-free interval J of size $c_{\text{dam-clear-s}} B$. If we had $J \nsubseteq I$ then the head would have escaped, contrary to the assumption; so we have $J \subseteq I$. Also, J is not merged with another hole, since this possibility already was excluded above. Therefore J is a new hole.

The definition in Lemma 12.2 implies $c_{\text{dam-free-esc}} n^2 \geqslant c_{\text{dam-clear-t}} n$. Then $t_{i+1} \leq t_i + c_{\text{dam-free-esc}} n^2 T$ for each i, and the sum of sizes of all the holes increases by at least B from t_i to step t_{i+2}.

□

12.3 Healing

In this section we will show how the healing procedure can eliminate islands.

The main new complexity compared to the corresponding analyses in [1] (what is called "healing" here was called "recovery" there) is the presence of damage. In [1], whenever healing started with an alarm, the procedure was brought to its conclusion as long as no new burst occurred. Now this is no more the case. Every time the head emerges from a damaged area, its state is arbitrary (the trajectory properties do not allow any conclusion about it). The trajectory properties will allow the elimination of damage eventually; but we will just rely on the many zigzags both in normal and in healing mode to make sure that any other change will be limited before this elimination.

First we consider the case with the fewest possible distractions of the main issue of damage elimination: no new burst, no other island nearby, and clearly no possibility for the head to leave without completed healing. Even in this case, we treat the elimination of the damage in first.

Lemma 12.6. *Suppose that in an annotated history, following a distress-free time interval, at some time t_0 an island I is encountered at a head position $x_0 = \xi.\mathrm{pos}$ at the front, further the following holds.*

(i) *No burst occurs during the following time interval of size $c_1\beta^3 T$, where c_1 will be obtained from the analysis.* ⟪*Peter: Determine it!*⟫

(ii) *There is no other island within distance $2E$ nearby.*

(iii) $x_0 = \chi.\mathrm{front}$*, that is the island is encountered when the head is moving forward in its sweep.*

Then the following holds.

(a) *The annotation of the history can be extended to a certain time $t' \le c_1\beta^3 T$ in such a way that the healing area will contain no more damage.*

(b) *The island and the healing area during all this is contained in a common interval of size $c_2\beta B$, where*

$$c_2 = (2c_{\mathrm{island}} + 1)<Z/2.$$

(c) *By the time t', in most cells, only the fields belonging to healing or rebuilding will be changed, not the fields belonging to the main simulation. The exceptions are cells of island and possibly β more cells adjacent to it.*

The condition that the island is encountered at the front excludes only the possibility that it is encountered after the sweep is turned back, and the backward zig beyond the colony end encounters an island. (This case will be handled later.)

Proof. At the beginning of the process, the healing area $H(t)$ coincides with the island I. We will indicate below how it changes. Whenever a cell gets marked for healing or rebuilding, the healing area will be extended over the cell. But it will also be extended in some other cases.
Let $D(t_0) \subseteq I$ be the smallest interval enclosing the damage in island I at time t_0. We will define for each subsequent time t an interval $D(t)$ enclosing the damage that is inherited from $D(t_0)$, and will find a time t with $D(t) = \emptyset$. The set $H(t) \setminus D(t)$ is a disjoint union of two intervals. Let $R(t)$ denote the interval on the right. We will look at the times that the head spends on the right of the damage. The same arguments apply to the left.
According to the Attacking Damage property of a trajectory in Definition 6.5, every time the head enters damage and then leaves it again to a distance

$c_{\text{dam-dist-2}}B$, damage will recede at least by an amount B. We will show that within a certain period this happens enough times to consume the whole damage. Let $\overline{D}(t)$ be the damage interval extended by $c_{\text{dam-dist-2}}B$ on both sides. Suppose that the head enters damage at a certain time t'. Then by the Lemma 12.2 (Damage-free escape), it has to leave $\overline{D}(t')$ within time $O(\beta^2 T)$. When it leaves on a side where it entered any time earlier, the damage will be decreased by the amount B. By enough repeated enterings and escapes, damage will be consumed.

The head cannot stay longer than $O(\beta^2 T)$ in the set $R(t)$, again due to the escape lemma. It may leave on the side of the damage, leading to the decrease of damage when it emerges again. Or, it may leave on the other side: we should see that in this case, it returns soon without increasing $H(t)$ much. For concreteness we look at the right side of $H(t)$. We would like to claim that every one-cell increase of this right side is compensated by an attack on the damage, but this is not exactly true. The exception is that the healing procedure can extend the healing area by a length $O(\beta B)$. But, as we will see, this can happen essentially only once. Our analysis will also show claim (c).

Call an interval I ***special***, if it has been marked by the marking part of the healing procedure or the rebuild procedure in one or several right sweeps (a later right sweep extending the marks further), after time t_0. The markings assume a starting point of healing or rebuilding to the left of I. We will call that the damage-elimination process arrived at a ***standard stage*** on the right, if either the damage is empty or the head is in the damage and the left end of $R(t)$ is covered by a special interval.

1. Suppose that we are at a standard stage, and the head leaves the damage on the right. Then the head moves back into the damage within $O(\beta)$ steps, leaving the process again in a standard stage, and increasing the healing area at most by $2B$.

   Proof. If the mode is normal, alarm is called immediately. This starts a new healing, just overwriting the left end of $R(t)$ by a new standard interval, and moving back to the left, into the damage.

   If the mode is healing or rebuilding then alarm is not called only if the head enters the left end cell of $R(t)$ exactly in a stage that allows for the extension of the standard interval in continuation of the marking part of the healing or rebuilding procedure. In this case, alarm will not be called while this procedure continues, since what is seen on the right is also what is expected. The

marking continues forward one step, then the head switches back again, and reaches damage. In all these cases, the healing area increases by at most one cell (a distance $<2B$) for a decrease of the damage size by B.

2. Let t_1 be the first time when the head steps to the right of the healing area, onto a cell y. Then after $O(\beta^2)$ steps spent on the right of $D(t)$, we arrive at a standard stage.

   If in the process n cells outside I had some field modified that is not a healing or rebuilding field, then these cells are adjacent to I and the damage decreased by at least nB. The healing area increases by at most $2c_{\text{island}}\beta B$.

   Proof. We distinguish cases depending on the mode at time t_1.

   2.1. Suppose that the mode is normal.

   If alarm is not called immediately, then it will not be called while the head is to the right of y. The head will either be sweeping to the right or to the left.

   Suppose it is sweeping to the right: then it zigs back immediately from the front in y. If alarm is called then the marking stage of alarm is entered, and as the marks spread, the head will enter the damage, creating a standard stage. This increases the healing area by at most one (not necessarily adjacent) cell for each cell encountered in the island along the way to the damage, so at most by $2c_{\text{island}}\beta B$.

   Whenever alarm occurs below, the analysis continues in this way, so we will be content to note that alarm occurred.

   If alarm is not called then the head reaches the damage in $O(\beta)$ steps, finding all cells along the way consistent with y. When it comes out of the damage again, if it comes out in a state doing marking for healing or rebuilding, then a standard stage will be created. When it comes out in another state, it may call alarm immediately. Every time it does not, the front on the right can repeatedly be advanced by at most one cell, before zigging back to reach damage again. If this was repeated n times before reaching a standard stage, then at most n cells had their non-healing, non-rebuilding fields modified: the ones to which the head advanced from y to the right.

   Suppose now that the head is sweeping to the left: then it is zigging to the right (without changing anything on the right of y). As long as alarm is not called, each cell the head steps onto is consistent with the healthy con-

figuration on the right and can be added to it, removing it from $H(t)$. For the head to reach the damage may take $O(\beta^2)$ steps due to the zigs. Only unmarked cells are left behind; from here on the above analysis applies to every time the head emerges from the damage.

2.2. Suppose that the mode is healing.

If the state is after the marking stage, then alarm is called immediately, since during this part of healing, the head should only step on marked cells. If the state is in the marking stage then either alarm is called immediately, or the marking task will be continued, creating a special interval I_1. Alarm may still be called if no marks are found on the left where they are expected: in this case, the left of I_1 may be overwritten by a special interval I_2, leading to a standard stage again. The maximum increase of the healing area is again at most $2c_{\text{island}}\beta B$.

2.3. Suppose that the mode is rebuilding.

The sweep can be left or right. If the sweep is left then alarm will be called in one or two steps: if not in the first step, then after zigging back and seeing that the right neighbor of y is not marked for rebuilding. If the sweep is right then the head zigs back immediately, and either calls alarm somewhere along the way or reaches the damage in $O(\beta)$ steps, leading to a standard stage. The maximum increase of the healing area is again at most $2c_{\text{island}}\beta B$.

For the estimate of part (b) of the lemma: the above analysis shows that marking on the right can add at most $2c_{\text{island}}\beta B$ to the size. It can add at most the same amount on the left as well. We had additional $n \le \beta$ increases due to n forward steps (left or right) that were followed by a backward zig: this leads to the estimate of the lemma. □

Now we consider the whole healing process. The lemma below has the same assumptions as the lemma above, the form of its conclusions is also similar, but it goes much farther, eliminating the island.

Lemma 12.7. *Suppose that in an annotated history, following a distress-free time interval, at some time t_0 an island I is encountered at a head position $x_0 = \xi.\text{pos}$ at the front, further the following holds.*

(i) *No burst occurs during the following time interval of size $c_3\beta^3 T$, where c_3 will be obtained from the analysis.* ⟪*Peter: Determine it!*⟫

(ii) *There is no other island within distance* $2E$ *nearby.*

(iii) $x_0 = \chi$.front, *that is the island is encountered when the head is moving forward in its sweep.*

Then the following holds.

(a) *The annotation of the history can be extended to a certain time* $t'' \le c_3\beta^3 T$ *in such a way that the island and the healing area disappear.*

(b) *The island and the healing area during all this is contained in a common interval of size* $c_4\beta B$, *where* $c_4 < Z/2$ *will be obtained from the analysis.* ⟪Peter: Determine it!⟫

(c) *By the time* t'', *in most cells, only the fields belonging to healing or rebuilding will be changed, not the fields belonging to the main simulation. The exceptions are cells of island and possibly* β *more cells adjacent to it.*

Proof. As the assumptions are stronger than for Lemma 12.6, we can draw that lemma's conclusions. Thus, the annotation can be extended to some time $t' \le c_1\beta^3 T$ when the damage disappears. When this happens the head is in or next to the healing area, which at this point has a size $\le c_2\beta B$. Due to our assumption of the lemma, the island is originally encountered by the front. At time t' the front is still inside the healing area, since we added to that area in part 2.1 of the proof in a way to avoid moving the front outside.

1. Suppose that alarm is called in $H(t')$ or in a cell neighboring it.

   Now the healing procedure has an undisturbed run. We add all marked cells to the healing area at the time of their marking. The procedure can carry out its corrections as shown in its description. Once it removed its marks, we delete the healing area and the island.

2. Assume that at time t' the mode is normal.

   In whichever direction the head starts moving, it may discover incoordination and call alarm, in which case the analysis of part 1 applies. If it does not alarm in the area $H(t')$ or immediately next to it, then it does not alarm farther from the area either. ⟪Peter: Elaborate!⟫ And as long as alarm is not called then due to zigging, it will cross the whole area $H(t')$ in at most $O(\beta^2 T)$ steps. Once this happens we can erase the island and the healing area, declaring the whole configuration normal. ⟪Peter: Elaborate!⟫

3. Assume that the mode is not normal.

   Suppose that the mode is healing.

If we are in the marking stage then the marking may continue, or alarm may be called. If alarm is called then it is called, either in $H(t')$ or next to it, in which case the analysis of part 1 applies. If alarm is not called then the healing will finish successfully.

If we are in a later stage then, unless this is the last sweep of the healing procedure, it will be discovered in one sweep that the area marked is too small: alarm will be called, with analysis as above. In the last sweep it may happen that the mode returns to normal: then the analysis of part 2 applies.

Suppose that the mode is rebuilding. The analysis is similar to the case of the healing mode, except that the marking stage will also always lead to alarm, since its zigging will always find that there is not a large enough interval already marked for rebuilding (by some earlier failed healing).

□

13 The simulation codes

Let us now define formally the codes φ_{*k}, Φ^*_k that are needed for the simulation of history $(\eta^{k+1}, Noise^{(k+1)})$ by history $(\eta^k, Noise^{(k)})$. Omitting the index k we will write φ_*, Φ^*. To compute the configuration encoding φ_* we proceed first as done in Section 6.3, using the code ψ_* there, and then add some initializations: In cells of the base colony and its left neighbor colony, the *cSw* and *cDrift* fields are set to Last(+1) − 1, 1, and Last(+1), 1 respectively. In the right neighbor colony, these values are Last(−1) and −1 respectively. In all other cells, these values are empty. The *cAddr* fields of each colony are filled properly: the *cAddr* of the j cell of a colony is j mod B^*. ⟪Peter: Picture?⟫

The value $Noise^{(k+1)}$ is obtained by a residue operation just as in Definition 5.2 of sparsity. It remains to define $\eta^* = \eta^{(k+1)}$ when $\eta = \eta^k$. Over a history that has been super admissible locally for a while, if no colony has its starting point at x at time t, set $\eta^*(x,t) = \text{Vac}$. Otherwise $\eta^*(x,t)$ will be decoded from the *cInfo* track of this colony, at the beginning of its work period containing time t. In the non-super admissible case, $\eta^*(x,t) = Bad^*$, contributing to the set $Damage^*$. More precisely:

Definition 13.1 (Damage scale-up)**.** Let $(\eta, Noise)$ be a history of M, where $Noise = Noise^{(k)}$ as in Definition 5.2. We define $(\eta^*, Noise^*) = \Phi^*(\eta, Noise)$ as follows. Let $Noise^* = Noise^{(k+1)}$. Consider position x, and let $I = [x - 2QB, x + 3QB)$, $J = (t - T^*, t]$. If the history $(\eta, Noise)$ has not been super admissible

on $I \times J$ then $\eta^*(x,t) = Bad^*$; assume now that it is, and let $\chi(\cdot,u)$ be some healthy configuration satisfying η over I at time u. If x is not a start of a colony $C = [x, x + QB)$ in $\chi(\cdot,t)$ then let $\eta^*(x,t) = \text{Vac}$; assume now that it is. Then let $t' \in J$ be the starting time in χ of the work period of C containing t, and let $\eta^*(x,t)$ be the value decoded from $\eta(C,t')$. In more detail, as said at the end of Section 5.2, we remove *PadLen* symbols from both ends of the track $\eta(C,t').cInfo$, and apply the decoding ψ^* to it to obtain $\eta(x,t)$. ⌟

Bibliography

[1] Ilir Çapuni and Peter Gács. A Turing machine resisting isolated bursts of faults. *Chicago Journal of Theoretical Computer Science*, 2013. See also in arXiv:1203.1335. Extended abstract appeared in SOFSEM 2012. 2, 1, 12.1, 12.3

[2] Bruno Durand, Andrei Romashchenko, and Alexander Shen. Fixed-point tile sets and their applications. *Journal of Computer and System Sciences*, 78(3):731–764, 2012. 2.2

[3] Peter Gács. Reliable computation with cellular automata. *Journal of Computer System Science*, 32(1):15–78, February 1986. Conference version at STOC' 83. 2.2

[4] Peter Gács. Reliable cellular automata with self-organization. *Journal of Statistical Physics*, 103(1/2):45–267, April 2001. See also arXiv:math/0003117 [math.PR] and the proceedings of STOC '97. 2.2, 2.5

[5] G. L. Kurdyumov. An example of a nonergodic homogenous one-dimensional random medium with positive transition probabilities. *Soviet Mathematical Doklady*, 19(1):211–214, 1978. 2.2